UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FABIO PETRILLO

Swarm Debugging: the Collective Debugging Intelligence of the Crowd

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Marcelo Soares Pimenta
Coadvisor: Profa. Dra. Carla M. Dal Sasso Freitas

Porto Alegre
September 2016

CIP — CATALOGING-IN-PUBLICATION

Petrillo, Fabio

Swarm Debugging: the Collective Debugging Intelligence of the Crowd / Fabio Petrillo. – Porto Alegre: PPGC da UFRGS, 2016.

125 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2016. Advisor: Marcelo Soares Pimenta; Coadvisor: Carla M. Dal Sasso Freitas.

1. Interactive debugging. 2. Crowd software engineering. 3. Software maintenance. 4. Software engineering. I. Pimenta, Marcelo Soares. II. Freitas, Carla M. Dal Sasso. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Oppermann
Vice-Reitor: Prof. Jane Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretor do Instituto de Informática: Prof. Luis da Cunha Lamb
Coordenador do PPGC: Prof. Luigi Carro
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"To Andreia and Isabelle."

"If debugging is the process of removing software bugs,
then programming must be the process of putting them in."

— EDSGER W. DIJKSTRA

ACKNOWLEDGEMENT

Firstly, my particular thanks to my wife Andreia for sharing all the moments and helping me to improve my peace and self-control.

All my gratitude to Professors Marcelo Pimenta and Carla Freitas for his trust and encouragement over the last years, and especially for their unconditional support, valuable advice, and commitment to guiding me through my doctoral research. Despite challenges and unusual situations, they ever workaround the problems to give an incredible opportunity. Moreover, probably their most important advising was not "correct" any time; I was free to make mistakes (and I made a lot of errors). Because of their approach, today I am stronger and prepared for any challenge.

I do not have words to express my gratitude to Professor Yann-Gaël Guéhéneuc. He received me as an invited researcher in his Lab at École Polytechnique de Montréal, and he was a mentor, a partner, a leader. He never demanded me anything, and he ever gives me a lot. It is an honor to work and collaborate with Professor Guéhéneuc, an illuminated human being. Another very essential person was Professor Foutse Kohn who advised me several times, shared great ideas and improved our research papers.

Several friends collaborate a lot to this work, specially Zéphyrin Soh, Guilherme Lacerda, Gabriel Veras, and Maurício Aniche. Thanks guys for your friendliness and support.

I would like to express my deep gratitude to my mother, Sonia. She gave me, with her simplicity, an incredible gift saying just one sentence on my first primary school day: "Go! You can go alone". She never demanded anything about my studies; she always trusted me. Now, I became the first Ph.D. in my family.

One Sunday morning, when I was leaving my home to work on this thesis, my little six-years-old daughter Isabelle gave me a firm hug and said to me: "Dad you will do, you will finish your work, I'm sure!". Certainly, my daughter, it was the most important encouragement and why we are here: because of you. I love you so much *my baby*!

ABSTRACT

Ants are fascinating creatures that beyond the advances in biology have also inspired research on information theory. In particular, their study resulted in the creation of the Information Foraging Theory, which describes how agents forages for information in their environment. This theory also explains recent and fruitful phenomena, such as crowdsourcing.

Many activities in software engineering have applied crowdsourcing, including development, translation, and testing, but one action seems to resist: debugging. Developers know that debugging can require dedication, effort, long hours of work, sometimes for changing one line of code only.

We introduce the concept of Swarm Debugging, to bring crowdsourcing to the activity of debugging. Through crowdsourcing, we aim at helping developers by capitalizing on their dedication, effort, and long hours of work to ease debugging activities of their peers or theirs, on other bugs.

We show that swarm debugging requires a particular approach to collect relevant information, and we describe the Swarm Debugging Infrastructure. We also show that swarm debugging minimizes developers effort. We conclude with the advantages and current limitations of swarm debugging and suggest directions to overcome these limitations and further the adoption of crowdsourcing for debugging activities.

Keywords: Interactive debugging. Crowd software engineering. Software maintenance. Software engineering.

Depuração em enxame: a inteligência coletiva na depuração pela multidão

RESUMO

As formigas são criaturas fascinantes que, além dos avanços na biologia também inspiraram pesquisas sobre teoria da informação. Em particular, o estudo resultou na criação da Teoria da Forragem de Informação, que descreve como os agentes de buscam informações em seu ambiente. Esta teoria também explica fenômenos recentes e bem-sucedidos, como crowd sourcing.
Crowdsourcing tem sido aplicado a muitas atividades em engenharia de software, incluindo desenvolvimento, tradução e testes, mas uma atividade parece resistir: depuração. No entanto, os desenvolvedores sabem que a depuração pode exigir dedicação, esforço, longas horas de trabalho, por vezes, para mudar uma linha de código único.
Nós introduzimos o conceito de Depuração em Enxame, para trazer crowd sourcing para a atividade de depuração. Através de crowd sourcing, pretendemos ajudar os desenvolvedores, capitalizando a sua dedicação, esforço e longas horas de trabalho para facilitar atividades de depuração.
Mostramos que a depuração enxame requer uma abordagem específica para recolher informações relevantes, e descrevemos sua infra-estrutura. Mostramos também que a depuração em enxame pode reduzir o esforço desenvolvedores. Concluímos com as vantagens e limitações atuais de depuração enxame, e sugerir caminhos para superar estas limitações e ainda mais a adoção de crowd sourcing para atividades de depuração.

Palavras-chave: depuração interativa, engenharia de software em multidão, manutenção de software, engenharia de software.

LIST OF ABBREVIATIONS AND ACRONYMS

SD	Swarm Debugging
SDI	Swarm Debug Infrastructure
GV	Debug Global View
SDT	Swarm Debug Tracer
IFT	Information Foraging Theory
SI	Swarm Intelligence
IDE	Integrated Development Environment
JPDA	Java Platform Debugging Architecture
SDS	Swarm Debug Services
SQL	Structured Query Language

LIST OF FIGURES

Figure 2.1 Visualization of dynamic call relations on an execution of JEdit, Java JRE classes using 3D-HEB. 44
Figure 2.2 Ant Colony....45
Figure 2.3 Views of a real world graph in R^3 (left, upper right) and R^2 (lower right) representing relations within a software system. The graph contains 1539 vertices, 1847 edges, and 126 clusters. (BALZER; DEUSSEN, 2007)....45
Figure 2.4 Ant Colony....46

Figure 3.1 Main concepts as foundations of Swarm Debugging 50
Figure 3.2 Swarm Debugging overview....52
Figure 3.3 The Swarm Debugging meta-model....54

Figure 4.1 The Swarm Tracer architecture....58
Figure 4.2 The Swarm Manager view....58
Figure 4.3 The Swarm Debug Services - architecture 60
Figure 4.4 The Swarm Debug metadata....61
Figure 4.5 Swarm Debug SQL Console....61
Figure 4.6 Swarm Debug Dashboard....62
Figure 4.7 Neo4J Browser - a Cypher query example....63
Figure 4.8 Sequence stack diagram for Bridge design pattern 64
Figure 4.9 Method call graph for Bridge design pattern....64
Figure 4.10 GV elements. A: types (nodes); B: invocations (edges); C: task filter area.65
Figure 4.11 GV showing data from all tasks with JabRef. 66
Figure 4.12 Breakpoint search tool (fuzzy search example)....66

Figure 5.1 Invocations (Dev/Task) by Elapse Time 77
Figure 5.2 Invocations (Dev/Task) by Elapse Time 78
Figure 5.3 Relation between time of first breakpoint and task elapsed time 79
Figure 5.4 Examples of fuzzy debugging patterns - Task 1 80
Figure 5.5 Examples of fuzzy debugging patterns - Task 2 80
Figure 5.6 Examples of straight debugging pattern - Task 1 81
Figure 5.7 Examples of straight debugging pattern - Task 2 82
Figure 5.8 Breakpoints by kind of statement - call, control flow and assignment....86
Figure 5.9 Methods with 5 or more breakpoints....88
Figure 5.10 Java expertise....92
Figure 5.11 GV by Task 0318....94
Figure 5.12 GV by Task 0667....95
Figure 5.13 GV by Task 0669....96
Figure 5.14 GV by Task 1026....97
Figure 5.15 GV by Task 1235....98
Figure 5.16 GV usefulness - experimental phase one....98
Figure 5.17 GV usefulness - experimental phase two....99
Figure 5.18 Video on Dev520 using GV to support the task 993. 99

LIST OF TABLES

Table 4.1 Illustration of co-breakpoint 68
Table 4.2 Approach for breakpoint prediction 69

Table 5.1 Elapse time by task (average) 75
Table 5.2 Breakpoints by kind of statement 86
Table 5.3 Breakpoints in the same line of code by task 87
Table 5.4 Breakpoints by type in different tasks 88
Table 5.5 Breakpoint Prediction Results 89

CONTENTS

1 INTRODUCTION **19**
1.1 Motivation **21**
1.1.1 Some scenarios and challenges 21
1.1.2 Towards Swarm Debugging 23
1.2 Objective and Contributions **24**
1.3 Research Questions **25**
1.4 Outline of the thesis **25**
2 BACKGROUND **27**
2.1 Program Comprehension **27**
2.2 Static and Dynamic Analysis of Software **30**
2.3 Data frugality **31**
2.4 Re-opened Bugs **31**
2.5 Debugging **31**
2.5.1 Interactive Debugging 33
2.5.2 Debugging Tools 34
2.5.3 Advanced Debugging Approaches 36
2.6 Information Foraging Theory **37**
2.7 Collective Behaviour and Swarm Intelligence **39**
2.8 Crowd on Software Engineering **46**
2.9 Final remarks **47**
3 SWARM DEBUGGING **49**
3.1 Foundations of Swarm Debugging **49**
3.2 Swarm Debugging overview **51**
3.3 Swarm Debugging meta-model **53**
3.4 Final remarks **55**
4 SWARM DEBUG INFRASTRUCTURE **57**
4.1 Swarm Debug Tracer **57**
4.2 Swarm Debug Services **59**
4.2.1 Swarm RESTful API 59
4.2.2 SQL Console 60
4.2.3 Full-text Search Engine 60
4.2.4 Dashboard Service 62
4.2.5 Graph querying console 62
4.3 Swarm Debug Views **62**
4.3.1 Sequence stack diagram 63
4.3.2 Dynamic method call graphs 63
4.3.3 Debug Global View 64
4.3.4 Breakpoint search tool 66
4.3.5 Starting/Ending method search tool 66
4.4 Definition of Co-Breakpoint **67**
4.5 Breakpoint Prediction **68**
4.6 Use scenarios **69**
4.7 Final remarks **70**
5 EVALUATION OF THE SWARM DEBUGGING **73**
5.1 Experiment 1 - towards understanding interactive debugging **73**
5.1.1 Context 74
5.1.2 Study Design 74

5.1.3 RQ1: Is there a correlation between the numbers of invocations and tasks' elapsed time? 76
5.1.4 RQ2: Is there a relationship between the number of breakpoints and tasks' elapsed time? 77
5.1.5 RQ3: Do developers explore/debug in different ways a task? 78
5.1.6 RQ4: Is there a correlation between the numbers of breakpoints and developers' expertise? 78
5.1.7 RQ5: Is there an association between time of first breakpoint and task's elapsed time? 79
5.1.8 Threats of Validity 81
5.2 Experiment 2 - mining debugging data to recommend breakpoints: an empirical study 82
5.2.1 Experiment setup 83
5.2.2 RQ1: How much time do developers spend to toggle the first breakpoint? 85
5.2.3 RQ2: On what kind of statement do developers toggle their breakpoints? 86
5.2.4 RQ3: Do developers toggle breakpoints in the same place? 87
5.2.5 RQ4: How effective is co-breakpoint for breakpoint prediction? 88
5.2.6 Results and Discussions 89
5.2.7 Threats to Validity 90
5.3 Experiment 3 - supporting maintenance tasks using shared debugging visualisations 91
5.3.1 Experiment design 91
5.3.2 Experiment setup 91
5.3.3 Is Global View useful to support software maintenance tasks? 94
5.3.4 Does sharing and visualizing debug data support software maintenance tasks? ... 95
5.3.5 Participants' Feedback 95
5.3.6 General Feedback 100
5.3.7 Threats to Validity 101
5.4 Final remarks 102
6 CONCLUSION 105
6.1 Summary of contributions 106
6.2 Limitations 107
6.3 Future work 108
REFERENCES 109
APPENDIX 117

1 INTRODUCTION

Developing and maintaining software is currently an incredibly complex activity (BOOCH, 2015). Software is an abstract artifact, virtual, intangible and difficult to understand (BROOKS, 1987; KNIGHT; MUNRO, 2000), especially when its size increases. The production and maintenance of software nowadays is mostly supported by means usage of Integrated Development Environments (IDEs) dominate the production and maintenance of software. In fact, 97% of .NET developers use Microsoft Visual Studio, and 73% of Java developers use Eclipse-based IDEs (GU, 2012). Besides, developers interact intensively with their IDEs while working on software maintenance.

In software maintenance, debugging is an everyday action (TANENBAUM; BENSON, 1973) during which developers use debugging tools to detect, locate and correct faults. Debugging is an activity of finding and fixing defects that prevent proper operations of software programs (IEEE, 1990). In fact, any process that aims at finding faults can be considered "debugging". The software engineering community usually focuses on one kind of particular debugging process, which consists in using a tool called *debugger*: interactive debugging.

Developers spend over two-thirds of their time (68%) investigating code, and the majority of their time is debugging (33%) (LATOZA; MYERS, 2010), frequently using debuggers in IDEs like Eclipse. However, software debugging is an arduous task that requires time, effort, and a good understanding of the source code (ZHENG et al., 2006), and developers use debugging tools to support these piece of work.

Debugging tools, *a.k.a. debuggers*, such as *sdb* (KATSO, 1979), *dbx* (LINTON, 1990), or *gdb* (STALLMAN; SHEBS, 2002) have been used by developers for decades. Modern debuggers are often integrated into development environments, *e.g.,* DDD (P. Wainwright, 2010) or the debuggers of Eclipse, Netbeans, Intellij IDEA, Visual Studio Integrated Development Environments (IDEs). In order to start a new interactive debugging session, developers must define where to toggle a useful breakpoint. Breakpoints are key for interactive debugging, and an important breakpoint is the first one during a session.

With debuggers, developers navigate through the system code, looking for locations to place breakpoints, and stepping into statements. While stepping, developers can traverse method invocations, toggle one or–more breakpoints, stop and–or restart execution. This exploration process allows developers to gain knowledge about programs and

the causes of faults, allowing them to fix them. Although debuggers have evolved since their inception in the sixties, developers' debugging activities have mostly remained the same.

During the past 30 years, the research community strove to provide developers with automated debugging tools. Automated debugging tools require both successful and failed runs and do not support programs with interactive inputs (KO, 2006). Consequently, developers have not adopted them in practice (PARNIN; ORSO, 2011). Moreover, automated debugging approaches are often unable to indicate the “right" locations of faults (RÖSSLER, 2012). Furthermore, recent studies showed that empirical evidence of the usefulness of many automated debugging techniques is limited (PARNIN; ORSO, 2011). Other more interactive approaches, such as slicing and query languages, help developers but, to date, there has been no evidence that they significantly ease developers' debugging activities.

The understanding of debugging activities could help practitioners and researchers to develop a new family of debugging tools that are more efficient and–or more adapted to the particularity of each debugging task. Moreover, assessing whether developers follow debugging patterns could be the first step towards recommending locations to toggle breakpoints that would reduce debugging effort and thus improve developers productivity.

In terms of productivity, ants are fascinating creatures that beyond the advances in biology have also inspired research on information theory. In particular, their study resulted in the creation of the Information Foraging Theory, which describes how agents forages for information in their environment. This theory also explains recent and fruitful phenomena, such as crowdsourcing.

Many activities in software engineering have applied crowdsourcing, including development, translation, and testing, but one action seems to resist: debugging. Developers know that debugging can require dedication, effort, long hours of work, sometimes for changing one line of code only.

We introduce the concept of Swarm Debugging, to bring crowdsourcing to the activity of debugging. Through crowdsourcing, we aim at helping developers by capitalizing on their dedication, effort, and long hours of work to ease debugging activities of their peers or theirs, on other bugs.

1.1 Motivation

Software engineering has several challenges. We designed five cases to illustrate some situations that often occurs.These cases are some examples that have inspired and motivated our work.

1.1.1 Some scenarios and challenges

Case 1: Recurrent and reopened bugs Jenifer works for a software company. In our regular workday, as a senior maintainer, she receives several times the same bug, because usually bug description is incomplete and incorrect. Consequently, the bug comes back from testing team each time with a new issue to solve. Moreover, frequently the reopen bug does not be solved by one developer that worked on previously, and she has to redo a full investigation process before solving the reopen bug. In this case:

- How Jenifer could use a previous knowledge from colleagues for improving her job?
- What were the used breakpoints during past sessions of debugging?
- What were the visited types during the previous sessions of debugging?

Case 2: Discovering software project's architecture John is an engineer who was hired to maintain a critical system written in Java. Today is his first job day, receiving some "welcome" instructions and a mission: to describe how the system operates and what its architecture. Furthermore, no one in the company knows the system. A person who maintained it won the lottery and suddenly left the company. He had built the software alone and had not left any documentation describing how it works or its organization. In fact, Jonh has only the source code to do his job. In this case:

- What John should do to understand the project?
- What kind of tools does John should use to do the job?
- What are the steps that John should take to find out the system's architecture?
- What is a starting point that John should look for doing the job?

Case 3: Looking for a maintenance area Peter is a junior programmer. He started very early to program and after completing a technical course on software development,

he was hired as an intern by a large software company. Your boss appointed him a seemingly simple task: to modify labels in user interfaces and reports. It seemed trivial to be done. However, the problem was more complex. He should change different systems with several technologies, and he had just a source code. Also, after a quick analyze, he figures out that the label was dynamically generated by a complicated module on which Peter debugged for hours to take on what context the labels were made by the system. In this case:

- Where Peter should toggle breakpoints to understand the complex module?
- What tools should he use to do the job?

Case 4: TDD, unit testing and code coverage Joseph has used unit testing and TDD for several years, getting good results. He also realized that software with a high percentage of unit testing coverage is safer to be modified by developers. Thus, Joseph would implement unit tests on a legacy project on which he is in charge now. As usual, the software has no implemented test unit class or documentation. In this case:

- to achieve a high coverage level, where Joseph should start implementing some tests?
- How could he find the best classes for implementing unit tests?
- What tools it could use to simplify their code analysis?

Case 5: Analyzing an open source project Mary is an experienced programmer who always learned and was inspired by open source projects. Thus, she decided to contribute to her favorite project, correcting a registered defect in its issue tracking tool. Browsing over some items, Mary chose one that believed to be simple for solving. She checked out the project source code and started to explore it in your favorite IDE. After more than three hours studying the project, Mary was frustrated: she can not understand how those thousands of classes work together to perform the software. In this case:

- Have you ever been in a similar situation?
- What could be done to help Mary to understand the software? What would you do in her place?
- What steps could she take to understand the project?
- What tools, models or techniques could she use?

1.1.2 Towards Swarm Debugging

An important aspect of current software engineering is that new developers expect collaborations (STOREY et al., 2014): the newer generation of developers is proficient in social media, for communication and learning. They are open, transparent, and hope to share their knowledge. Hence, a debugging tool that collects and shares knowledge about debugging sessions among developers is likely to gain developers' support and adhesion.

Roßler (RÖSSLER, 2012) advocates for the development of a new family of debugging tools that are context-aware and that rely on exact scenarios (even though Ceccato *et al.* (CECCATO et al., 2015) recently showed that automatically-generated test-cases are as useful for debugging as manual test cases). To build context-aware debugging tools, researchers need an understanding of developers' debugging activities and the contextual factors of fault fixing activities. Thus, researchers need tools to collect and share data about developers' interactive debugging activities.

In a recent study about program comprehension, Maalej *et al.* (MAALEJ et al., 2014) found that developers follow practical understanding strategies depending on context. They try to avoid understanding as possible and often put themselves in the role of users by inspecting graphical interfaces. Usually, developers interact with the application user interface to test whether the application behaves as expected and to **find starting points** for further inspection. Moreover, developers need to acquire runtime information, and debuggers are frequently used by developers to comprehend programs and obtain information (MURPHY; KERSTEN; FINDLATER, 2006) (MAALEJ et al., 2014). Finally, developers use **transient notes** as comprehension support. This **externalized knowledge is only used personally**. It is neither archived nor reused.

Moreover, Maleej*et al.* identified a gap between the state of the art and the state of the practice in program comprehension. Context information also appears to perform an essential role when providing the knowledge needs of developers in program comprehension scenarios. Unfortunately, their findings reveal that **context information is typically implicit and not captured**.

Maleej*et al.* suggests that the implementation of such tools would require **the instrumentation of the IDE and continuous observation of the work of developers**. To this end, they found that **developers are willing to share information about their work collected by the IDE automatically, the vast majority of developers agree to share non-personal information, such as artifacts used and experience made, while**

comprehending software. These trends should be further studied and considered when designing and introducing new context-aware, personalized methodologies and tools for program comprehension, including accessing and sharing knowledge about programs.

Moreover, developers must find suitable breakpoints when working with debuggers, establishing a starting point to debug. However, to set adequate breakpoints, deciding where to toggle them, is a highly laborious task (TIARKS; RÖHM, 2012), and often developers prefer simplistically toggle several irrelevant breakpoints. This practice causes an overhead and usually a waste of time and effort.

Thus, if developers spend a lot of time debugging code, why this human effort is lost? Could we collect debugging session information to (re)use in the future, creating visualizations and searching tools about this information? Further, more particularly, why is a breakpoint toggled in a class? What is its purpose?

1.2 Objective and Contributions

Our principal objective is to address these debugging issues and opportunities, claiming that **collecting and sharing debugging session information can provide data to create new visualizations and searching tools to support programmer tasks, decreasing the time for developers deciding where to toggle the breakpoints and improve the project comprehension**. To support this statement, we propose a new approach named Swarm Debugging .

Swarm Debugging is an approach that use collective intelligence reasoning to collect and share interactive debugging data, providing visualizations and searching tools to support software maintenance activities.

This thesis presents two main contributions. First, we introduce the concept of Swarm Debugging, to bring crowdsourcing to the activity of debugging. Through crowdsourcing, we aim at helping developers by capitalizing on their dedication, effort, and long hours of work to ease debugging activities of their peers or theirs, on other bugs. Second, to help research studies on debugging and, thus, help improving our understanding of how developers debug systems using debuggers, we propose the Swarm Debug Infrastructure (SDI), with which practitioners and researchers can collect and share data about developers' interactive debugging activities. In order to evaluate the effectiveness of the Swarm Debugging , we have conducted experiments that aims to understand how developers apply interactive debugging and answering our research questions. As a consequence, some

results and findings are an indirect contribution towards debugging phenomenon comprehension.

1.3 Research Questions

This thesis has several research questions:

- Is there a correlation between the numbers of invocations and tasks' elapsed time?
- Is there a relationship between the number of breakpoints and tasks' elapsed time?
- Do developers explore/debug in different ways a task?
- Is there a correlation between the numbers of breakpoints and developers' expertise?
- Is there an association between time of first breakpoint and task's elapsed time?
- How much time do developers spend to toggle the first breakpoint?
- On what kind of statement do developers toggle their breakpoints?
- Do developers toggle breakpoints in the same place?
- Do developers toggle breakpoints in the same place?
- How effective is co-breakpoint for breakpoint prediction?
- Is Global View useful to support software maintenance tasks?
- Does sharing and visualizing debug data support software maintenance tasks?

1.4 Outline of the thesis

We organized the remainder of this work as follows:

- **Chapter 2** summarizes some essential concepts for understanding our approach also describing some related work concerning not only "debugging tools" and "interactive debugging", but software visualisation, software static and dynamic analysis, release changes and re-opened bugs' studies, foraging information theory, and collective intelligence as well;
- **Chapter 3** describes the main aspects of our proposed approach: the Swarm Debugging;
- **Chapter 4** present the SDI, as defined to support the SD approach;
- **Chapter 5** evaluates our approach by presenting and discussing results on applying in three reported experiments; and

- **Chapter 6** presents the final considerations of this thesis, including the contributions, challenges, and future work.

2 BACKGROUND

This section provides a conceptual background for supporting our approach. First, we discuss (1) program comprehension (finding starting method, locating and recommending relevant program elements to developers). Next, (2) re-open bugs, (3) debugging, (4) static and dynamic analysis of software, (5) Information Foraging Theory, (6) collective behavior and swarm intelligence, and finally (7) crowd on Software Engineering.

2.1 Program Comprehension

Previous work studied how developers understand programs and provided tools to support program understanding. Maalej *et al.* (MAALEJ et al., 2014) observed and surveyed developers to comprehend how they investigate programs. They reported that to understand a program, developers need to acquire runtime information and frequently execute the application using a debugger. Ko *et al.* (KO, 2006) observed that developers spend large amounts of times navigating between program elements. They modeled program understanding as a process of searching, relating, and collecting relevant information. Sillito *et al.* (SILLITO; MURPHY; De Volder, 2008) identified the questions that developers ask when finding and extending starting methods. They described how developers answer these questions during software maintenance activities.

Feature and fault location tools are used to identify and recommend program elements that are relevant to a task at hand (WANG; LO, 2014). These tools are used to recommend relevant pieces of code to developers, the bug report (ZHOU; ZHANG; LO, 2012), domain knowledge (YE; BUNESCU; LIU, 2014), version history and bug report similarity (WANG; LO, 2014). In contrast to these approaches, Mylyn uses developers' activities (interaction traces collected during maintenance tasks) to reduce developers information overhead and to show in the developers' IDE only program elements that may be relevant to the task (KERSTEN; MURPHY, 2006). Mylyn interaction traces have been used to study work interruption (SANCHEZ; ROBBES; GONZALEZ, 2015), editing patterns (YING; ROBILLARD, 2011), and program exploration patterns (SOH et al., 2013). In addition to Mylyn, other tools collect data during maintenance tasks.

In a recent study about program comprehension, Maalej *et al.* (MAALEJ et al., 2014) found that developers follow practical comprehension strategies depending on con-

text. Developers try to avoid understanding whenever possible and often put themselves in the position of users by inspecting interfaces. Participants affirmed that standards, experience, and personal communication promote understanding. Their results reveal a gap between research and practice, as they did not observe any use of comprehension tools and developers seem to be unaware of them, finding a call for reconsidering the research agendas **towards context-aware support**.

Usually, developers interact with the application user interface to test whether the application behaves as expected and to **find starting points** for further inspection. Moreover, developers need to acquire run-time information, and debuggers are frequently used by developers to comprehend programs and obtain information (MURPHY; KERSTEN; FINDLATER, 2006) (MAALEJ et al., 2014). Other significant findings from Maleej's study (MAALEJ et al., 2014) is that developers' experience plays a major role in program comprehension activities and helps to identify starting points for further inspection and to filter out code locations that are irrelevant for the current task. Finally, developers use **transient notes** as comprehension support. This **externalized knowledge is only used personally**. It is neither archived nor reused.

Maleej *et al.* identified a gap between the state of the art and the state of the practice in program comprehension. State-of-the-art program comprehension tools were either unknown or rarely utilized even by experienced developers. Consequently, research effort should be focused on how research results can be used to support developers in their everyday work and on how software comprehension features can be integrated into the workflows depending on the task. Furthermore, context information also seems to play an important role when supplying the knowledge needs of developers in program comprehension scenarios. Unfortunately, their findings reveal that **context information is typically implicit and not captured**.

Moreover, Maleej *et al.* suggest that the implementation of such tools would require **the instrumentation of the IDE and continuous observation of the work of developers**. To this end, they found that **developers are willing to share information about their work collected by the IDE automatically, the vast majority of developers agree to share non-personal information, such as artifacts used and experience made, while comprehending software**. Finally, one of their most interesting findings is that developers put themselves in the role of users at the start of a comprehension process and try to collect and comprehend usage data. They observed developers inspecting a visible behavior in user interfaces and comparing it to expected behavior in bug fixing

as well as in other implementation tasks. This strategy aims at understanding the program behavior and structure, and at getting first hints for further program exploration, an alternative to reading and debugging source code. This observation suggests the potentials of usage data at deployment time to facilitate comprehension at development time. These trends should be further studied and considered when designing and introducing new context-aware, personalized methodologies, and tools for program comprehension, including accessing and sharing knowledge about programs.

Parnin *et al.* (PARNIN; ORSO, 2011), investigated aspects of automating debugging tools, observed that participants might be using automating debugging tools to find other statements that are near the fault, but ranked higher than the fault. They suggest that **programmers may use tools to identify starting points** for their investigation, some of which may be near the fault. Parnin *et al.* (PARNIN; ORSO, 2011) determined that programmers do not visit each statement in a linear fashion, and the **navigation pattern was not linear**. They claim that only giving the statement was not enough for the participants to understand the problem and that more context was needed, which made us conclude that complete bug understanding is not a realistic assumption. In the same study, Parnin *et al.* (PARNIN; ORSO, 2011) reported that the primary benefit of automation tools was **to point them in the right direction**. They would not necessarily use the tool to find the exact faulty statement, but **rather identified several candidate causes for the failure**. Even if the tool did not point the exact place of the fault, displaying **relevant starting points could help program understanding considerably**, and debugging tools may be more successful if they focused on searching through or **automatically highlighting certain suspicious statements**. Besides, some participants wanted more **program structure information** with the statements (e.g., method names). They also **wanted alternative views**, different ways of sorting and, in general, different ways of interacting with the data.

Parnin *et al.* (PARNIN; ORSO, 2011) claim that for a tool to be useful, it should **seamlessly integrate the distinct parts of the debugging technique** considered and provide end-to-end support for it. Although some of these issues can be addressed with careful engineering, it may be useful to focus research efforts on ways to **streamline and integrate activities** such as coverage collection, test-case grouping, and code review.

Finally, LaToza and Myers (LATOZA; MYERS, 2010) suggested that tools could significantly improve developer productivity by supporting searches across possible paths for statements matching a wider variety of search criteria, and their results also suggest

that developers could perform coding tasks more quickly and accurately with tools that more directly support answering reachability questions.

2.2 Static and Dynamic Analysis of Software

Static analyses examine a piece of code in the absence of input data and without running the code (AYEWAH et al., 2008). It could identify potential security violations (SQL injection), runtime errors (dereferencing a null pointer *e.g.,*) and logical discrepancies (a conditional test that can't possibly be true). In most circumstances, the analysis is performed on a particular version of the source code, and in the other cases, on several software versions.

Dynamic analysis is the analysis of data gathered from a running program, providing a picture of software behavior (CORNELISSEN et al., 2009). It typically comprises the analysis of a system's execution through interpretation (e.g., using the Virtual Machine in Java) or instrumentation, after which the resulting data are used for such purposes as reverse engineering and debugging. Several studies about dynamic analysis have to be highlighted as (CORNELISSEN et al., 2009),(CORNELISSEN; ZAIDMAN; DEURSEN, 2011), (GRATI; SAHRAOUI; POULIN, 2010), (ZIADI et al., 2011), and (LABICHE; KOLBAH; MEHRFARD, 2013). Dynamic analysis brings some benefits that are (CORNELISSEN et al., 2009):

- The precision concerning the actual behavior of the software system, for example, in the context of object-oriented software with its late binding mechanism.
- The fact that a goal-oriented strategy can be used, which entails the definition of an execution scenario such that only the parts of interest of the software system are analyzed.

Some limitations that can be highlighted are (CORNELISSEN et al., 2009):

- inherent incompleteness of dynamic analysis, as the behaviors or traces under analysis capture only a **small fraction of the usually infinite execution domain** of the program under study. Note that the same limitation applies to software testing.
- **difficulty of determining which scenarios to execute to trigger the program elements of interest**. In practice, test suites can be used, or recorded executions involving user interaction with the system.
- **scalability** of dynamic analysis due to the **large amounts of data** that may be

introduced in dynamic analysis, affecting performance, storage, and the **cognitive load**

2.3 Data frugality

Datensparsamkeit or *data frugality* (FOWLER, 2016) is an attitude to how we capture and store data, saying that we should only handle data that we need.

2.4 Re-opened Bugs

In large software development projects, when a programmer is assigned a bug to fix, she typically spends a lot of time searching (in an ad-hoc manner) for instances from the past where similar bugs have been debugged, analyzed and resolved (ASHOK et al., 2009). **No single person understands all the code in such systems**, and often new hires, who have little or no context about the code, are given the job of debugging failures. A recent study (BUGDE et al., 2008) revealed that developers and testers spend **significant time** during diagnosis looking for **similar issues that have been resolved in the past**. For example, the same bug or a very similar bug may have been encountered and fixed in another code branch, and programmers would greatly benefit from knowing this information.

According to Nguyen *et al.* (NGUYEN et al., 2010), a bug-fixing change is considered recurring if it is repeated identically or with relevant, slight modifications on several code fragments at one or multiple revisions. Kim *et al.* (KIM; PAN; WHITEHEAD, 2006) confirmed the existence of recurring bugs, showing that there are **about 20-40% of total fixing changes appear repeatedly**.

Finally, An *et al.* (AN; KHOMH; ADAMS, 2014), who investigated reopened faults in open-source systems, report that reopened faults can account for up to 10% of all reported faults in a system. Thus, we argue that debugging knowledge should be saved and shared among developers. Software development is, in general, a cooperative effort (FUGGETTA, 2000).

2.5 Debugging

> ***Debug.*** *To detect, locate, and correct faults in a computer program. Techniques include the use of breakpoints, desk checking, dumps, inspection, reversible execution, single-step operations, and traces.*

—IEEE Standard Glossary of SE Terminology—

Debugging is the process of discovering and fixing faults that prevent correct operations of software programs (IEEE, 1990). Thus, any process that aims at finding defects can be considered “debugging”. The software engineering community usually focuses on one kind of particular debugging process, which consists in using a tool called *debugger*: interactive debugging.

Developers have used many processes and tools to debug programs. These processes and tools range from control flow graphs, profiling tools, logs (BLASCIAK; PARETS,), to static analyses (BALL; RAJAMANI, 2002).

Understanding the cause of a failure typically involves complex tasks, such as browse on program dependencies and execute a piece of software several times with different inputs (PARNIN; ORSO, 2011). When a failure occurs, developers must perform three main essential activities. First, fault localization that consists of identifying the program statement(s) responsible for the failure. Second, fault understanding involves comprehending the reason for the failure. Indeed, fault correction is determining how to modify the code to remove it. Briefly, fault localization, understanding, and correction are associated with debugging.

Software debugging methods can be divided into fault localization and fault correction techniques (HOFER; WOTAWA, 2012). Fault localization methods focus on narrowing down potential faulty areas. They include spectrum-based fault localization, delta debugging, program slicing, and model-based software debugging.

Araki *et al.* described the debugging activity as an interactive process where developers make hypotheses, then verify them by examining the problems (ARAKI; FURUKAWA; CHENG, 1991). Developers then refute or validate their hypotheses until the tasks are resolved. Katz and Anderson (KATZ; ANDERSON, 1987) conducted several experiments to study how developers debug programs. They observed that the participant understanding is affected by their debugging behavior. The participant ability to fix faults is not affected by their debugging behavior, but participants faced difficulties locating faulty program elements.

Zeller (ZELLER, 2006) introduced the concept of scientific debugging, claiming that general debugging process could be inspired by the scientific method. Given an issue report, developer tries reproducing the bug, observing the failure. Next, based on the observation, developers create hypotheses about the cause of the failure. After that, developer tests their hypotheses. To do that, developers can print a value at a particular

location to determine whether to reject the hypothesis or not. Often, they may change a source code and evaluate whether the program performs as expected. If the program gives an unexpected result, it means the hypothesis is rejected. If the hypothesis is rejected, a developer needs to make a new hypothesis about the failure cause. If the hypothesis is supported and the bug is fixed, programmers stop the debugging process. If the hypothesis is supported, but the bug is not fixed, developers refine the hypothesis and test it again.

Debugging is a time-consuming activity, taking as much as 50% of programming time (ZAYOUR; HAMDAR, 2016). Developers spend prolonged periods of times in debugging sessions to find faults or to understand a program (LATOZA; MYERS, 2010). During these sessions, using traditional debugging tools, they gather a lot of information and create mental models of the program (MURPHY; KERSTEN; FINDLATER, 2006; STOREY, 2006). Unfortunately, several studies have shown that developers quickly forget details when they explore a different location in the source code, losing parts of their mental models (TIARKS; RÖHM, 2012). In a recent research, Tiarks and Röhms (TIARKS; RÖHM, 2012), who investigated the behavior of 28 professional developers, observed that to recall parts of their mental models, some developers write notes on pieces of papers or use external editors as short-term memory.

Finally, developers must find suitable breakpoints when working with debuggers (TIARKS; RÖHM, 2012) to reproduce the data and control flow of the program. Tiarks and Röhms (TIARKS; RÖHM, 2012) observed that professional developers encounter problems to set adequate breakpoints and that deciding where to toggle breakpoints is an “extremely difficult" task. They also showed that, often, developers set a lot of breakpoints at the beginning of their debugging sessions to then discard irrelevant breakpoints while they debug the program, which causes a significant overhead to developers.

2.5.1 Interactive Debugging

Interactive debugging consists in using a debugger tool, yet known as *program animation*, *stepping*, or *following execution* (ZAYOUR; HAMDAR, 2016). Also, many developers refer to this process as simply *debugging*, because several IDEs provide debuggers to support *debugging*.

However, while *debugging* is the process of finding faults, *interactive debugging* is one particular debugging process in which developers use tools to investigate the execution of a program interactively. We use the expressions *interactive debugging* or *stepping*,

but there is not yet a consensus on what is the best name for this debugging process.

2.5.2 Debugging Tools

When maintaining and evolving programs, developers must locate and understand the causes of the faults. Some developers tend to print pieces of text (*e.g.,*`printf()` in C) to identify faulty program elements, but a debugging tool is essential in any programming environment (CHIC; NIERSTRASZ; GIRBA, 2013), as it helps developers to understand the dynamic behavior of software systems. Debugging tools have been developed to help developers locate and–or understand essential elements. Researchers have studied to improve debugger tools to address the developer issues. In this section, we review related research to Swarm Debugging. Our work transverses several domains, including debugging techniques, visual debugging, databases, and collective intelligence. This section highlights some debugging tools proposed by several researchers.

Hipikat (CUBRANIC; MURPHY, 2003)(CUBRANIC et al., 2005) is an Eclipse open-source tool that groups an implicit memory for a project by analyzing links between stored artifacts, and that recommends them from the archives that are relevant to a task. Through two qualitative studies, they have shown that this approach helps a newcomer perform a task adequately on an unknown system. It also provides search for code-related artifacts, basing on the notion of relationships among artifacts, which are derived from "project memory," which identifies paths traversed by earlier members of a team, and textual similarity. The textual similarity element is a hand-crafted textual match that specifies weights to words based on a global prevalence of the word in the repository and local prevalence of the word to the document. Also, Hipikat supports foraging by reducing the cost of navigation among code and non-code artifacts, such as bug reports, CVS logs, and e-mails (CUBRANIC et al., 2005). Our approach shares same perspective, but with build knowledge from several debugging sessions, while Hipikat's recommendations are made purely based on the content of artifacts (CUBRANIC et al., 2005).

Romero *et al.* (ROMERO et al., 2007) extended the work of Katz and Anderson (KATZ; ANDERSON, 1987) and identified high-level debugging strategies, *e.g.,* stepping and breaking execution paths and inspecting variable values. They reported that, according to their background and their level of expertise, developers use the information available in the debugging environment differently.

JIVE (GESTWICKI; JAYARAMAN, 2005; CZYZ; JAYARAMAN, 2007) is an

Eclipse plug-in that is based on the Java Platform Debugging Architecture (JPDA) and can analyze Java program executions. JIVE requests notification of some runtime events, including class preparations, step, variable modifications, method entries and exits, thread starts and ends. JIVE capture respectively, the current execution state and execution history of a Java program, providing two kinds of runtime visualizations of Java programs - object diagrams and sequence diagrams. However, the JIVE sequence diagrams can become long and unwieldy (JAYARAMAN; JAYARAMAN; LESSA, 2016), and Wang et al. (JAYARAMAN; JAYARAMAN; LESSA, 2016) proposed a new approach their compact representation.

DebugAdvisor (ASHOK et al., 2009) is a recommender system to improve debugging productivity by automating the search for similar issues from the past. Like Hipikat (CUBRANIC et al., 2005), DebugAdvisor includes past bug reports, logs of interactive debugger sessions, data on related source code changes, and data about people who can be consulted. These information sources are a mixture of structured and unstructured data, and ignore all the structure in the data, using only full-text search to index and search through the data. This approach has the advantage that could reuse existing indexing and searching tools; however, the experimental result showed that there is much room for improvement in the quality of retrieved results (ASHOK et al., 2009). Differently, of DebugAdvisor, we use structured and unstructured debugging data in our approach. Furthermore, DebugAdvisor does not use debugging fine-grained events to suggest breakpoints or explore debugged software regions. It is limited as an unstructured searching tool.

Debugging Canvas (DELINE et al., 2012) is a tool where developers debug their codes as a set of code bubbles, explained with call paths and variable values, on a two-dimensional pan-and-zoom surface. Code Bubbles employed a call stack bubble in the debugger to show the parent methods above the current bubble in the call stack bubble. DeLine *et al.* showed Code Bubbles is useful with long or complex code paths, with large code bases with many layers, with unfamiliar code bases, and when the code involved dynamically linked code, factories, or other indirect forms of control flow. SD uses the similar idea, but it integrates source code and visualization in two separate views.

Minelli *et al.* (MINELLI et al., 2014) presented a visual approach to understanding and characterise development sessions from the UI perspective, collecting, visualizing, and analyzing hundreds of development sessions and report on their findings. To collect data, they used DFlow, a tool that records all interactions with the IDE. Barr and Marron (BARR; MARRON, 2014) proposed *TARDIS*, a time-traveling debugger (TTD) that

supports reverse execution, helping to speed hypothesis formation and validation loop by allowing developers to navigate backward, as well as forwards, in a program's execution history. Moreover, Zayour and Hamdar (ZAYOUR; HAMDAR, 2016) studied the difficulties faced by developers when debugging in modern IDEs. They reported that the nature of the IDE affects the time spent by developers during debugging activities.

2.5.3 Advanced Debugging Approaches

Several advanced debugging approaches have been proposed for last years. First, Zheng *et al.* (ZHENG et al., 2006) presented a systematic approach to *statistical debugging* of software programs in the presence of multiple bugs, using probability inference and common voting framework to accommodate more general bugs and predicate settings.

Ko and Myers (KO, 2006; KO; MYERS, 2009) introduced the *interrogative debugging*. It is a paradigm that allows developers to ask questions directly about their programs' output, helping them to determine more efficiently and accurately what parts of the system to understand. They evaluated their paradigm using a prototype called the Whyline, and they claim that their approach reduces debugging time by a factor of eight.

Pothier and Tanter (POTHIER; TANTER, 2009) proposed *Omniscient debuggers*, and approach to support back in time navigation across previous program states. In omniscient debuggers, all events from the execution are logged and can be inspected and queried. As a result, programmers can ask the debugger which events (e.g., an assignment) are responsible for a current state (e.g., a variable being null). Performance and footprint are two issues in omniscient debugging because of **all events from program execution** must be processed and stored efficiently.

Delta debugging (HOFER; WOTAWA, 2012) is an approach proposed by Hofer *et al.* that the smaller the failure-inducing input, the less program code is covered. It can be used to minimize a failure-inducing input systematically.

Ressia *et al.* (RESSIA; BERGEL; NIERSTRASZ, 2012) proposed an alternative approach named *object-centric debugging*, focusing on objects as the key abstraction execution for many typical developer tasks. The central point of object-centric debugging is to allow users to perform operations directly on the objects in a computation, instead of performing operations on the execution stack. Object-centric debugging supports debugging operations that are specific of the Object Oriented paradigm (e.g., set a breakpoint on an object call). The premise of object-centric debugging is that a static representation

of the program (the code) and the stack-based run-time model adopted by most debuggers are not suitable for higher level abstractions - objects in the case of object-centric debugging. Further, object-centric debugging is based on a runtime representation of the program.

Estler *et al.* (ESTLER et al., 2013) discussed the *Collaborative Debugging* describing CDB, a debugging technique and integrated tool designed to support effective collaboration among developers during shared debugging sessions. Their study suggests that debugging collaboration features are perceived as important for developers, and can improve the experience in collaborative context. Differently of CDB (synchronous debugging tool), our approach uses asynchronous debugging sessions. Consequently, our approach does not have several collaboration issues founded on CDB.

Chen and Kim (CHEN; KIM, 2015) proposed mining the Stack Overflow QA site to leverage the large mass of crowd knowledge to aid developers while debugging a system. *Crowd Debugging* detects defective code and suggests solutions with explanations for developers. The authors perform crowd debugging in eight high quality and well-maintained projects, claiming that it can identify bugs that are invisible to existing tools.

Salvaneschi and Mezini (SALVANESCHI; MEZINI, 2016) presented the RP Debugging, a new debugging paradigm which provides support to inspect and reason about the flow of changes through a reactive programming application. When an application is debugged with RP Debugging, users can visualize dependency graphs and use them as a model for reasoning about the application execution, implementing this approach in Reactive Inspector, a debugger for reacting.

2.6 Information Foraging Theory

Information foraging theory (IFT) is founded on optimal foraging theory, developed by Pirolli and Card (PIROLLI; CARD, 1999) to understand how individuals search for information. Optimal foraging theory is inspired in the biological sciences, in investigations and theories of how animals hunt for food; this started with Pirolli and Card for finding relationships between users' information search patterns and animals' food foraging strategies.

In the field of information technology, the predator is the person in need of information, and the prey is the information itself (LAWRANCE et al., 2008). Using concepts such as "patch," "diet" and "scent," IFT describes the most suitable pages a web-site user

will visit in pursuit of their required information (prey), by clicking links carrying words that are a near match to (smell like) their information need. The scent of information comes from the semantic relationships between words expressing an information need and words contained in links to web pages.

Human “predators” searching for information “prey” look at many sources. They look for “scents” by following "cues" in the environment, indicators of the relation of information sources to prey. Inspired by appeals in the psychological literature for ecological accounts of contextually dependent human behaviors, information foraging theory offered a new perspective for evolving strategies to find information relevant to their needs efficiently **without processing everything, minimizing the mental cost** to achieve their goals.

Lawrance *et al.* (LAWRANCE; BELLAMY; BURNETT, 2007) reported a study that makes use of information foraging theory to analyze how professional developers explore on source code during maintenance. Their results showed that information foraging theory was a **significant predictor of the developers’ maintenance behavior** (LAWRANCE et al., 2008). They claim that information foraging theory can provide the foundations needed for tool development. Thus, they believe that programmers may be **information foragers when debugging** because their research has shown that when debugging, **programmers create hypotheses and then search for information to verify (or refute) this hypotheses**. Also, they conjectured that such hypotheses are linguistically related to the words in the bug reports, and consequently, bug reports define the programmer’s goal and the scent they are seeking, attempting to fix bugs found a correlation between the bug report and the set of classes visited by the programmers (LAWRANCE et al., 2008).

In another study, Kuttal *et al.* (KUTTAL; SARMA; ROTHERMEL, 2013) showed that the stronger scents available within mashup programming environments could improve users’ foraging success, leading to a new model for debugging activities framed concerning information foraging theory to support debugging.

Fleming *et al.* (FLEMING et al., 2013) studied the applicability of Information Foraging Theory for understanding information-intensive software engineering tasks. They concluded that **without environment support, foraging during debugging may be tedious and costly**, and in IFT terms, setting **breakpoints enriches the environment by creating low-cost links**. Finally, Piorkowski *et al.* (PIORKOWSKI; FLEMING, 2013) claim that diversity is essential regarding IFT. Even though developers were pursuing the

same overall goal (find the bug) during a debugging session, they sought highly diverse diets, suggesting that debugging tools need to support "long tail" demand curves of programmer information.

2.7 Collective Behaviour and Swarm Intelligence

Software systems are large and complex currently (HILL, 2010). To manage this complexity, software teams have used *collaborative and self-organisation* approaches (CHOW; CAO, 2008), and we claim that software projects have some analogies with collective behaviors. That is an important conceptual aspect of our approach, and we present several studies that explore this concept, applying the finding that many collective human behaviors (debug foraging *e.g.,*) are similar to their animal counterparts (SUMPTER, 2006).

In recent years, the notion of self-organization has been used to understand collective behavior of animals (SUMPTER, 2006). The central principle of self-organization is that simple **repeated interactions between individuals can produce complex adaptive patterns at the level of the group**. Inspiration comes from patterns seen in physical systems, such as spiraling chemical waves, which arise without complexity at the level of the individual units of which the system is composed. The suggestion is that biological structures such as termite mounds, ant trail networks and even **human crowds** can be explained concerning repeated interactions between the animals and their environment, without invoking individual complexity.

Sumpter (SUMPTER, 2006) argues that the key to understanding collective behavior is in identifying the principles of the behavioral algorithms followed by individuals and of how information flows between the animals. These principles, such as **positive feedback**, response thresholds, and individual integrity are repeatedly observed in very different animal societies. **The individual units do not have a complete picture** of their position in the overall structure and the structure they build has a form that extends well beyond that of the individual units.

A social insect colony is a superorganism, without a brain, and each worker has access to only very local information (DENEUBOURG et al., 1990). To understand how complex colonies are made without a central coordination, emerging from a collective behavior, Deneubourg *et al.* claim that it is necessary to use an approach in which the common pattern is seen as resulting from **autocatalytic interactions between simple**

and identical explorers. Moreover, this analogy may be made on the formation of paths, for example, by mammals in scrub or grassland. The more individuals have passed at one position, more vegetation is trampled, and less resistance it offers.

Studying ant colonies, Li *et al.* (LI et al., 2014) proposed that the entire foraging process of ants is guided by three successive strategies: hunting, homing, and path building, showing the **transition from chaotic to periodic regimes** observed in their model **results from an optimization scheme** for group animals with a home. According to that investigation, the behavior of such insects is not represented by random but **rather deterministic walks** (as generated by deterministic dynamical systems) in a random environment: the animals use their intelligence and experience to guide them. **The more knowledge an ant has, the higher its foraging efficiency is**. When young insects join the collective to forage with old and middle-aged ants, it benefits the whole colony in the long run. The resulting strategy can even be **optimal foraging**. Moreover, to survive, ants need to leave their nest and forage for food. The survival-of-the-fittest mechanism entails that ants do find not only food but also **an optimal path between their nest and the food source**, reflecting the **collective intelligence of the insects**. Nest and food source indeed play important roles in ants' foraging behavior.

For example, if an ant in a small colony finds a food source a long way from the nest, then by the time another ant passes over the place she left a pheromone trail, the pheromone will probably have evaporated (DENEUBOURG et al., 1990). In this case, the trail does not help other ants find the food. For large colonies of ants, however, it is more likely that an ant will cross the pheromone trail before it evaporates and reinforce it. The reinforcement leads to the familiar positive feedback loop and a well-established trail between nest and food. Thus, the output of the system, i.e. number of ants visiting the feeder and hence food collected, is low for small ant colonies but rapidly increases as the colony becomes larger. It is in this sense they call ant **foraging self-organized**: ants follow only local rules regarding the laying and following of pheromone, but the **resulting trail structure is built on a scale well beyond that of a single ant**.

Swarm Intelligence is a combination of simple agents interacting locally and with their environment, usually following very simple rules. Although there is no centralized control structure organizing how individual agents should behave, interactions between such agents lead to the emergence of global behavior, unknown previously by individual agents.

The roots of swarm intelligence are deeply embedded in the biological study

of self-organized behaviors in social insects (GARNIER; GAUTRAIS; THERAULAZ, 2007). Self-organization is a set of dynamical mechanisms whereby structures appear at the global level of a system from interactions among its lower-level components, without being explicitly coded at the individual level. It relies on four basic ingredients (GARNIER; GAUTRAIS; THERAULAZ, 2007):

1. **positive feedback**: results from the execution of simple behavioral "rules of thumb" that promote the creation of structures.
2. **negative feedback**: counterbalances positive feedback and that leads to the stabilization of the collective pattern, resulting from the limited number of available foragers, the food source exhaustion, and the evaporation of pheromone or competition between paths to attract foragers.
3. **stochastic actions**: randomness enables the colony to discover new solutions. For instance, lost foragers can find new, unexploited food sources, and then recruit nest mates to these food sources.
4. **stigmergic interactions**: self-organization requires multiple direct among individuals to produce apparently deterministic outcomes and the appearance of large and enduring structures.

Garnier *et al.* (GARNIER; GAUTRAIS; THERAULAZ, 2007) proposed that collective behaviors in social insects can be understood as the combination of coordination, cooperation, deliberation, and collaboration. Each of these functions emerges at the collective level from the unceasing interactions between the insects. Together, the four functions of organization produce solutions to the colony problems and may give the impression that the colony as a whole plans its work to achieve its objectives.

The increased flexibility of collective structures in an insect colony triggered by simple modulations of the individual behavior opens ways toward the design of self-adaptive swarm intelligent systems. The pursuance of experimental investigations in biological systems and the development of new theoretical frameworks about the adaptive role of these modulations should encourage the emergence of new applied studies.

Tschinkel (TSCHINKEL, 2015) suggested that the central mystery of social insect behavior is for everything they accomplish without a leader, without a blueprint, without a plan, without prior instruction, and in the case of subterranean ant nests, in the dark, but with a shared set of behavioral rules (TSCHINKEL, 2015). Each worker carries within herself the "instructions" of what to do given a certain context, how to interact with other

workers and the forming nest, and when to do it. The tasks are carried out in "series-parallel", that is, each worker does a small part of the job, after that, another worker may do the next step, while the first responds to something else. Applying a labor highly redundant, the work gets done in thousands of little steps, each seeming perhaps undirected, random or even backward, but statistically, progress is directional, and the nest gradually appears.

> *Under the right circumstances, groups are remarkably intelligent and are often better than the smartest person in them.*
> — James Surowiecki: *Wisdom of the Crowds During* —

Collective intelligence is shared or group intelligence that emerges from the collaboration, collective efforts, and competition of many individuals and appears in consensus decision making. The term appears in sociobiology, political science and context of mass peer review and crowdsourcing applications(WIKIPEDIA, 2015).

In the context of software engineering, according to Bruch *et al.* (BRUCH et al., 2010), today's Integrated Development Environments (IDEs) only integrate the tools and knowledge of **a single user and workstation**. This neglects the fact that the way in which we develop and maintain a piece of software and interact with our IDE provides a rich source of information that can **help ourselves and other programmers to avoid mistakes in the future**, or improve productivity otherwise. They argue that, shortly, IDEs will undergo a revolution that will significantly change the way in which we develop and maintain software, through the integration of collective intelligence, believing that there is great space for improvement by exploiting the knowledge of the masses. However, to unleashing the full power of the crowds, software engineering community has to provide an appropriate environment for building and evaluating IDE.

Bruch *et al.* (BRUCH et al., 2010) and Storey *et al.* (STOREY et al., 2014) claim that collective intelligence is an open-field for new software development tools. Bruch *et al.* (BRUCH et al., 2010) argue that actual Integrated Development Environments (IDEs) only integrate tools for and knowledge of a single developer and leave out other developers. Moreover, developers use IDEs only because they integrate all tools necessary to browse, manipulate, and build programs. If developers have questions about a particular piece of code, they must go outside of their IDEs to find answers, for example by asking colleagues or searching on-line. After they found an answer, the newly gained knowledge is *usually lost* inside the IDEs. Besides, Storey *et al.* (STOREY et al., 2014) argue that new developers expect collaborations: the newer generation of developers is proficient in social media, for communication and learning. They are opened, transparent, and expect

to share their knowledge.

Gu *et al.* (GU, 2012) defends that the next generation of IDEs has to incorporate a general framework to capture and exploit IDE interactions, and create an ecosystem of developer-aware applications and plugins. Additionally, we share their vision that the key to building the next generation of IDEs is to transform IDEs from being order-takers into intelligent, user-aware programs that monitor and reason about how their users interact with them.

The collective intelligence is inspired on a self-organization of ant societies. This self-organization could be observed on this quotation:

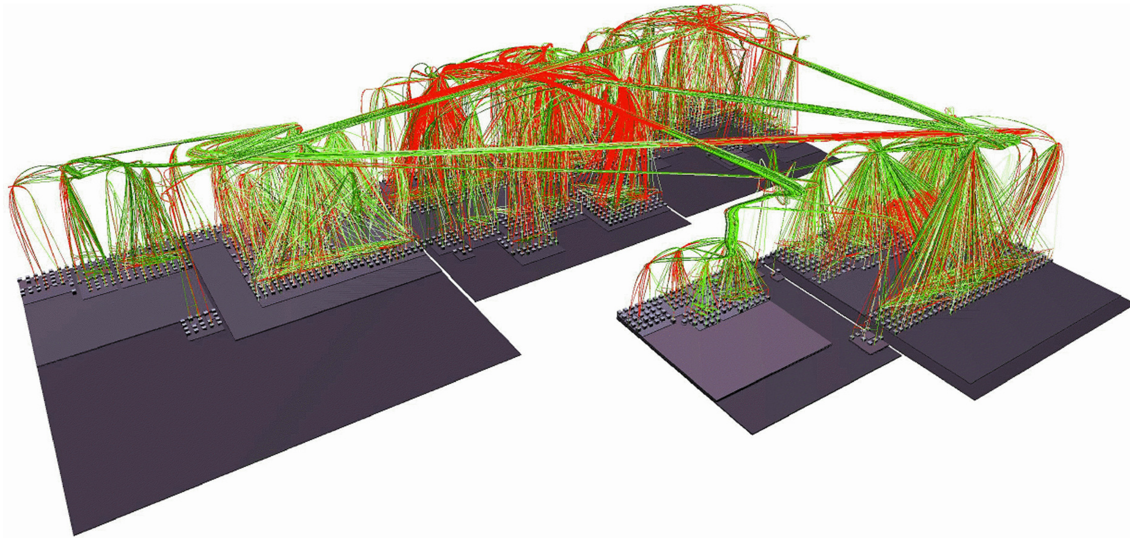
> *Studying self-organization in ants can make us aware that much of human activity is also self-organized, that is, that the tasks are accomplished by competent, distributed agents operating under a shared set of rules and interacting directly with the task itself. In humans as in ants, intense communication is not necessary to get the job done, and the outcome is to vary degrees an emergent phenomenon. This is apparent at the scale of economies, but also operates at many scales in human societies. It seems likely that we overestimate the importance of hierarchy and underestimate that of self-organization in human societies. Thus, we might profit from careful analysis of how leaderless groups of individuals with shared behavioral programs, be they ants or humans, interact to get the job done.*
> — Walter R. Tschinkel: *The architecture of subterranean ant nests: beauty and mystery underfoot* —

Like ant's societies (TSCHINKEL, 2015), developers usually does not have a global picture of a software system (BALL; EICK, 1996), and this brings us to another concept: emergence. Emergence involves local, and to some extent random, interactions between agents resulting in intelligent global behavior emerging on which hundreds of simple events can add up to make something complex happen. In nature, this global behavior is usually beyond the scope of understanding of the simple agents, so they are just doing what they are hard-wired, or ''programmed", to do. The complex, emergent behavior renders a selective advantage as simple organisms can punch well above their weight - a swarm of ants might repair a nest very quickly for example.

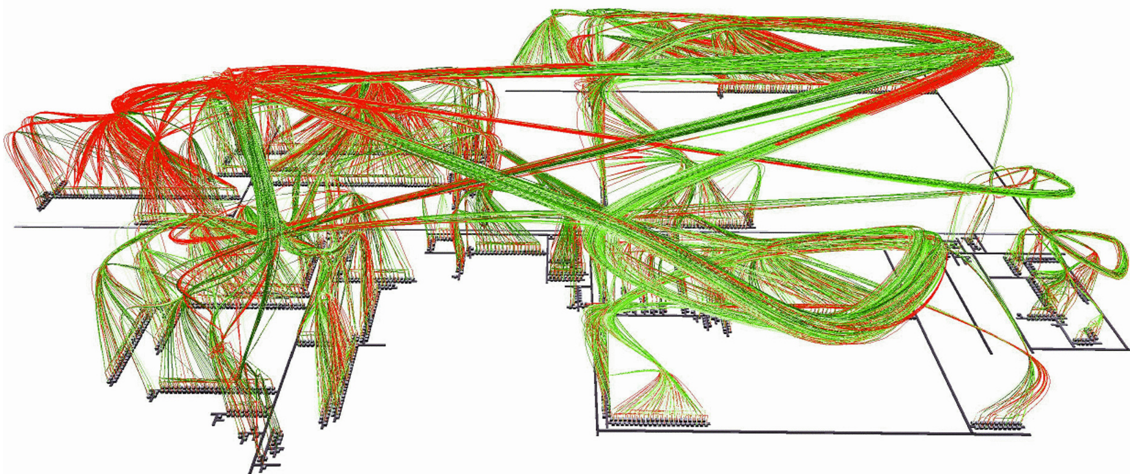
In fact, a metaphor of ant societies and software systems could also be analyzed in terms of architectural organization. For instance, Caserta *et al.* proposed a visualization using a city metaphor (Figure 2.1), we compare the visualization with an ant picture colony (Figure 2.2) presented in Tschinkel's work, observing some similarities. Another insightful example of similarities between ant nest and software could be observed in Figure 2.3 on a visualization proposed by Balzer *et al.* (BALZER; DEUSSEN, 2007) and a giant nest of ants showed by Hölldobler (HÖLLDOBLER; WILSON, 2009).

In fact, **this similarities between self-organization on ant nest and software**

Figure 2.1: Visualization of dynamic call relations on an execution of JEdit, Java JRE classes using 3D-HEB.



(a) Relations on top of the nested layout of the software city metaphor



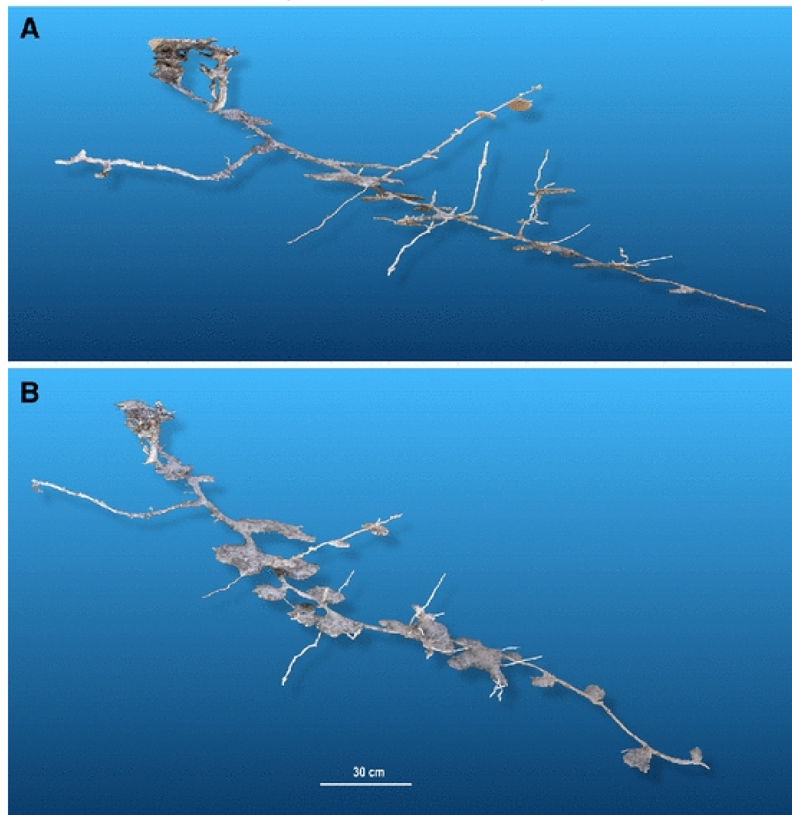
(b) Relations on top of the street layout of the software city metaphor

Source: (CASERTA; ZENDRA; BODéNèS, 2011)

architecture might not be a coincidence. Cockburn claims best architectures, requirements, and designs **emerge from self-organizing teams**, growing in steps and following the changing knowledge of the team and the changing wishes of the user community (COCKBURN, 2006)[1]. Nevertheless, these similarities are highlighted when Highsmith and Cockburn (HIGHSMITH; COCKBURN, 2001) discussed that software process development is a complex adaptive system and decentralized on which autonomous individuals interact to create innovative and **emergent results**.

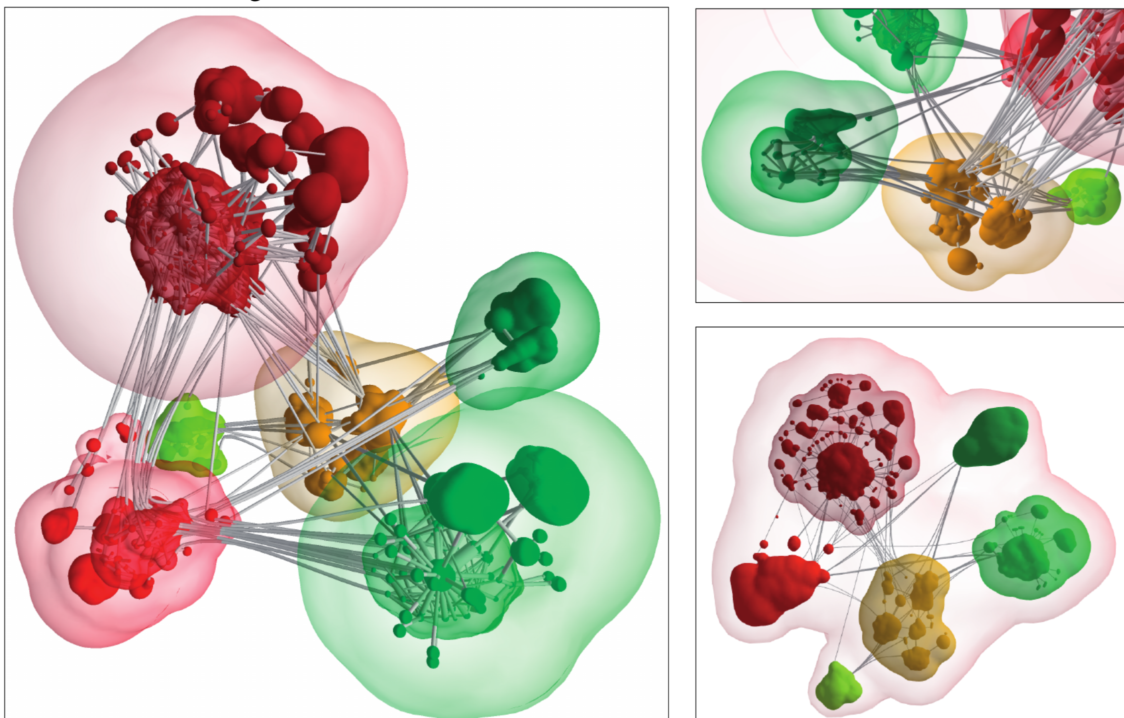
[1]These similarities open may be some opportunities to a new metaphorical approach for software organization: software could be represented like an ant nest architecture. However, we did not investigate that claims in this thesis, requiring future studies to evaluate its pertinence.

Figure 2.2: Ant Colony



Source: (TSCHINKEL, 2015)

Figure 2.3: Views of a real world graph in R^3 (left, upper right) and R^2 (lower right) representing relations within a software system. The graph contains 1539 vertices, 1847 edges, and 126 clusters. (BALZER; DEUSSEN, 2007)



Source: (BALZER; DEUSSEN, 2007)

Figure 2.4: Ant Colony

Source: (TSCHINKEL, 2015)

2.8 Crowd on Software Engineering

Crowd is a large group of people that are gathered or considered together (WIKIPEDIA, 2016). The term "the crowd" may sometimes refer to the so-called lower orders of people in general (the mob). A crowd may be definable through a common purpose or set of emotions, such as at a political rally, a sports event, or during looting (this is known as a psychological crowd), or may simply be made up of many people going about their business in a busy area.

In software engineering context, crowd development envisions a new way in which to build software, encompassing transient, fluid workforces automatically arranged by the environment to perform microtasks within a workflow (NADA, a). As in any potentially disruptive idea, it is far from clear in what contexts, if any, it may ultimately prove its value. But in exploring questions such as what context and information are required by developers in micro tasks, the exploration itself may create important new scientific knowledge about the nature of software development work, which may be broadly valuable in many ways.

In this context, Storey *et al.* (STOREY et al., 2014) advocate that the rise of the "social programmer" who actively cooperates in online communities and openly contributes to the production of a vast body of crowdsourced socio-technical meaning. The most recent generation of developers are using and expecting collaboration; **they are by nature open, transparent and assume to share**. They claim that the software development has entered a social era concerning of media use in software engineering. However, research into some of these relatively new phenomena indicates **a lack of tool support for crowdsourcing activities in software engineering** (STOREY et al., 2014). Finally, they elaborated several questions about collective intelligence and crowd software engineering: 1) What are the barriers that may block or turn some potential collaborators away? 2) What kinds of new tools can entice, improve and capture crowd-based participation in software engineering? 3) How do we ensure crowd diversity, as a “smart crowd” depends on having diverse skills and viewpoints?

2.9 Final remarks

Debugging is a time-consuming program activity. However, although the software engineering community provides advanced debugging approaches and tools to improve fault localization and program understanding, none of these collect debugging activities data to help on understanding of debugging activities with the goal of improving software debugging tools and–or program comprehension.

Unfortunately, several findings reveal that currently debugging context information is typically implicit and not captured, despite clearly to understand the program, developers need to acquire run-time information and frequently execute the application using a debugger. Consequently, researchers have observed that personalized methodologies and tools for program comprehension, including accessing and sharing knowledge about programs should be proposed and investigated.

Furthermore, several studies have highlighted how debugging could be improved applying Foraging Information Theory or a collective intelligence perspective. Finally, we discussed some aspects of collective behavior that open new avenues for debugging approaches.

In the next chapter, we present Swarm Debugging , our approach to addressing some of those opportunities towards context-aware support.

3 SWARM DEBUGGING

Swarm Debugging is an approach inspired by swarm intelligence (SI) and information foraging theory (IFT) to support and improve debugging activities. As we mentioned in Chapter 2, traditionally, debugging is an isolated activity performed by an individual, collecting information about a system intended to find possible fails. The SI and IFT concepts inspired us to think a different way of doing debugging. Using the debugging strategies adopted by several individuals to find fails (IFT approach) but as a super organism outcome from series of activities performed autonomously by team members (SI approach). Exactly as a swarm, our approach takes advantage of the same motivations underlying on crowd software engineering.

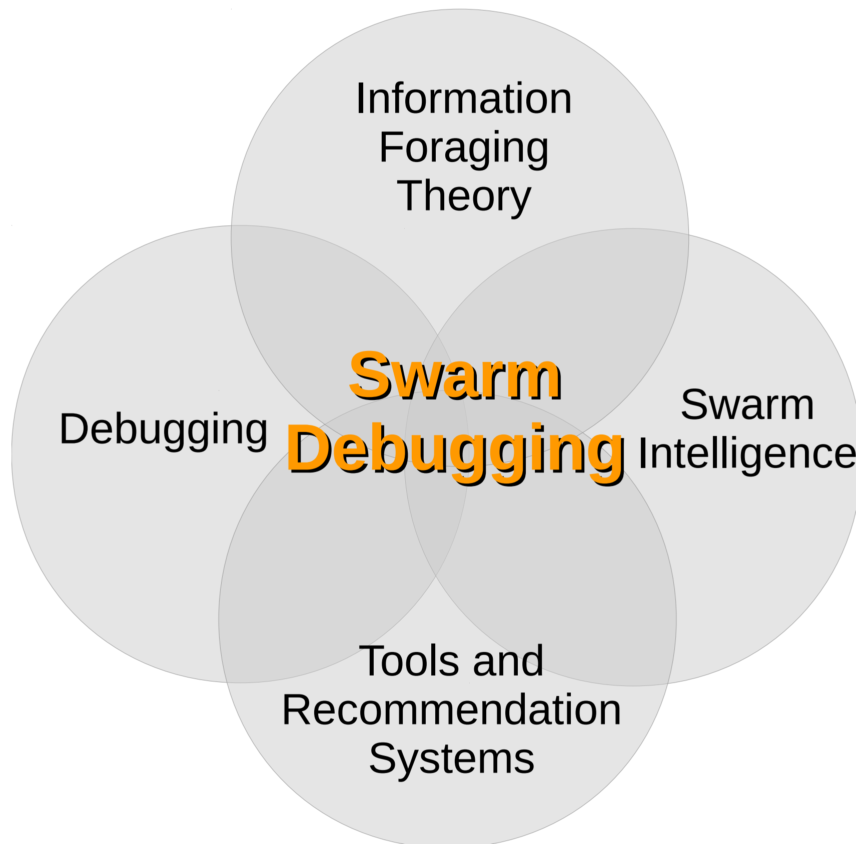
In this chapter, firstly we discuss the concepts that inspired this approach. Then, we describe key aspects of our approach. Finally, to represent the data that integrate the approach and explain their relations, we present a conceptual domain meta-model to describe Swarm Debugging dataset.

3.1 Foundations of Swarm Debugging

Swarm Debugging (SD) was inspired by Information Foraging Theory (IFT, Section 2.6), swarm intelligence (Section 2.7), and previous approaches (Sections 2.1 and 2.5.2), which formed a conceptual framework in which interactive debugging can be used to create this "intelligence" through the provided context-aware tools. Clearly, it should be noticed that these concepts are well known, but their integration for debugging is an original idea. In the following sections, we present how Swarm Debugging proposes this integration in a way that it is positioned at the intersection of these ideas (see Fig. 3.1).

Debugging is a foraging process in software systems (FLEMING et al., 2013; PIORKOWSKI; FLEMING, 2013), and we are not the first ones to consider IFT as useful for debugging (LAWRANCE et al., 2013). Indeed, these authors adopt IFT as a model **one prey/one predator** for debugging. Swarm Debugging uses these assumptions to support an **idea that interactive debugging environments could be used as an IFT environment**. However, differently from the IFT proposal that uses a model **one prey/one predator**, our approach extends this concept, applying **many developers on (different or a same) tasks working independently**, but sharing debugging traces to make knowledge and behaviors emerge, creating a swarm intelligence environment as a result.

Figure 3.1: Main concepts as foundations of Swarm Debugging



Source: from author

Thus, Swarm Debugging applies swarm intelligence concepts (Fig. 3.1) to support interactive debugging (refer to A in Fig. 3.2). It collects data during debugging sessions (refer to B in Fig. 3.2), storing data as breakpoints, reachable paths (refer to C in Fig. 3.2), transforming this data into knowledge through visualizations, searching tools, and recommendation systems (refer to D in Fig. 3.2). Finally, this knowledge about software projects is shared, producing a positive feedback loop (Fig. 3.2).

Briefly, exactly like ants in a swarm, developers work individually to build knowledge that **appears** to be orchestrated by some collective intelligence. In fact, the emphasis here is on *appears* because there is no particular orchestration. Individual actions are combined with the actions of others, and the collective intelligence emerges from this interaction.

Unfortunately, no previous work leverages the developers' swarm intelligence to improve debugging activities even though software development is, in general, a cooperative effort. This fact led Maale *et al.* (MAALEJ et al., 2014) to pose a question: *"Why don't share knowledge about programs?"*. They defend that the working context also influences developers' knowledge exchange behavior, which also depends on others, the

tasks, the size of the companies, and the number of collaborators. Swarm Debugging aims to fill this gap, designing and introducing new context-aware, personalized approach and tools for debugging and program comprehension, including accessing and sharing knowledge about programs.

Swarm Debugging complements other approaches proposed by *Mylyn* (Section 2.1), *Hipikat* and *DebugAdvisor* (Section 2.5.2), using debugging session data (either structured or unstructured), and analysing fine-grained events to collect breakpoints or information about debugged software areas. SD, Hipikat, and DebugAdvisor share the same essential idea: use previous data to support debugging tasks. Moreover, SD and Mylyn share the idea of considering context-awareness in their activities. In fact, SD and these approaches are complementary because each one treats different aspects of various debugging data. These previous works have not supported sharing knowledge about debugging sessions, especially with a context-aware perspective. Only the Swarm Debugging approach combines and shares multiple context-aware debugging sessions to explore breakpoints and interactive debugging paths in a collective way.

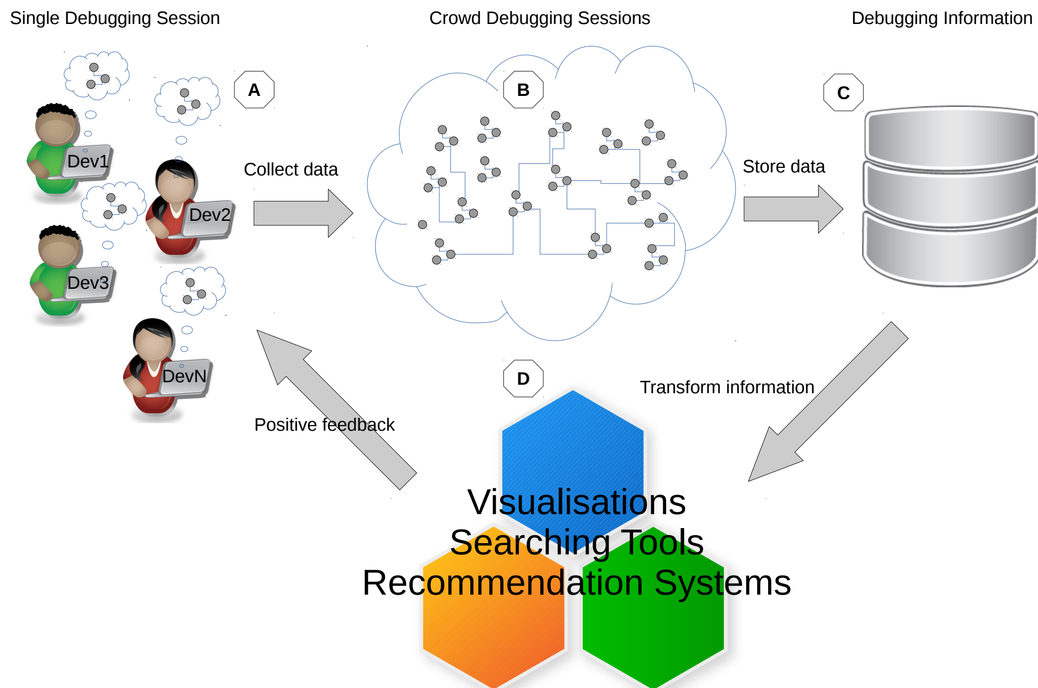
In fact, Swarm Debugging is a strategy to create an integrated, context-aware support for knowledge exchange. It enables the automatic capturing and sharing of knowledge with its context by observing developers' debugging interactions.

3.2 Swarm Debugging overview

Swarm Debugging (SD) is an approach that uses swarm intelligence applied on interactive debugging data to create knowledge to support software development activities. To achieve this goal, we propose SD as illustrated in Figure 3.2.

First, several developers perform their regular activities during interactive debugging sessions. Each debugging session begins by associating a task (i.e. an issue ticket) to this session. Each task is usually assigned to a developer who must perform debugging activities during a debugging session. For each single debugging session (Figure 3.2-A), events as toggling breakpoints and stepping events are collected transparently. All data produced by the developer's crowd are compiled (Figure 3.2-B) and stored in a debugging information repository (Figure 3.2-C). Finally, the information is transformed by mechanisms to create visualizations (using visualization tools), to mine debugging data (using searching tools) and recommend breakpoints (using a recommendation system)(Figure 3.2-D), providing an emerging knowledge for developers and closing a positive feedback

Figure 3.2: Swarm Debugging overview



Source: from author

loop.

This positive feedback is very important to SD. Using SD, developers produce knowledge about the software, which increases as time goes, and this knowledge created by the feedback is in fact accumulated. The original knowledge (A, on moment zero) is modified through all the processes of debugging (i.e., after all steps B, C and D) generating a new knowledge about the system. This positive feedback loop provides for each new debugging interaction a new state of knowledge about the software system, continuously growing until a complete coverage. Finally, this knowledge can be used by any developer, because it is shared and it supports both regular tasks (as *e.g.,*software comprehension - see Section 2.1) and re-opened bugs (see Section 2.4).

During a debugging session, developers analyze the code, toggling breakpoints and stepping through it. In traditional dynamic analysis approaches (Section 2.2), all interactions, states or events are collected by tools, typically tracing all data without any developer decision's control or context. Consequently, a huge quantity of data is collected by tracers, becoming both extremely hard and very costly to analyze this data volume. For using a typical dynamic analysis approach, developers have to install and run an intrusive infrastructure to collect data, which is not part of their regular tasks. Moreover, a traditional dynamic analysis usually has separate tools, adopting an approach *collect-all-mining-after* as a big data approach.

Swarm Debugging uses a data frugality approach (Section 2.3), collecting only paths that were **intentionally explored** by developers, collecting only methods **explicitly visited** by developers. It is a fundamental difference between SD and previous approaches. It means that we collect only invocations where developers call a method and intentional events (*e.g., Step Into* or F5 in Eclipse IDE) for visiting a method invoked by an analyzed method. Our approach permits that we inspect selected areas in a project because only visited paths (chains of invocations) are collected. Briefly, our approach typically collects only calls that are visited intentionally and analyses only these data, with less effort in a context-aware environment.

3.3 Swarm Debugging meta-model

We have defined domain concepts to model software projects and debugging data in our approach. This meta-model has two main goals. First, it represents the conceptual model of the SD approach. By definition, a conceptual model is a mapping of the concepts and relations of a domain, reflecting the real-world relationships and dependencies. Thus, the meta-model summarizes the central concepts adopted in SD. Second, it presents the essential elements necessary to build an infrastructure for SD.

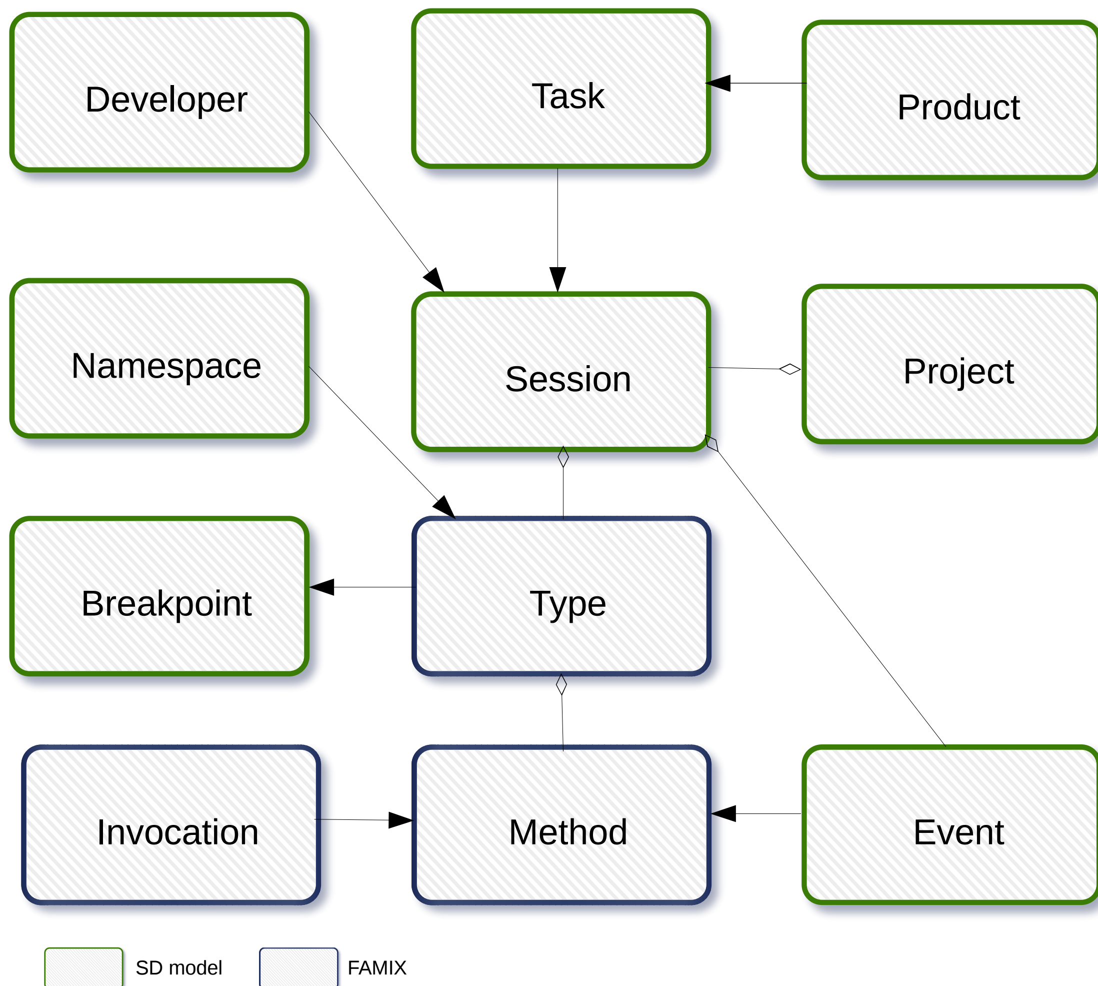
The concepts are inspired and complement the simplified FAMIX Data model(DEMEYER; DUCASSE; LANZA, 1999) with debugging data. FAMIX exploits meta-modelling techniques to make the data model extensible. The simplified view of the FAMIX data model comprises the main object-oriented concepts — namely Type, Method, plus the necessary associations between them — namely Invocation and Access.

The Swarm Debugging meta-model concepts (Fig. 3.3) include:

- **Developer** is a SD user, which creates and executes debugging sessions.
- **Product** is the target software product. Product is a set of source code projects (one or more).
- **Task** is a task to be executed by developers, like software comprehension, bug location, software maintenance or refactoring.
- **Session** represents a Swarm Debugging session. It relates developer, project, and debugging events.
- **Type** represents classes and interfaces in the project. Each type has a source code and a file. To simplify, SD only considers types that have source code available as belonging to the project domain, ignoring external libraries.

- **Method** is a method, procedure or function associated with a type, which can be invoked during debugging sessions.
- **Namespace** is a container for types. In Java, for example, namespaces are declared with the keyword *package*.
- **Invocation** is a method invoked from another method. It is formed by a pair of methods: an invoking (caller) and an invoked (called).
- **Breakpoint** represents the data collected when a developer toggles a breakpoint in an IDE. Each breakpoint is associated with a type and a method if appropriate.
- **Event** is an event data that is collected when a developer performs some actions during a debugging session, typically stepping events (step into a method, step over, run, return to the caller, *e.g.,*).

Figure 3.3: The Swarm Debugging meta-model



Source: from author

3.4 Final remarks

In this chapter, we presented the foundations of Swarm Debugging approach, how it works and its meta-model. Swarm Debugging provides a transparent approach for developers to share debugging information, creating a collective intelligence about their software project. In Chapter 4 we will present the Swarm Debugging Infrastructure aimed at supporting such debugging approach.

4 SWARM DEBUG INFRASTRUCTURE

Swarm Debugging (SD) is a conceptual approach that uses collective intelligence applied on debugging data. To evaluate SD, we built an infrastructure to support it: **Swarm Debug Infrastructure**. The Swarm Debug Infrastructure (SDI) implements the SD approach, providing a set of tools for collecting, storing, sharing, retrieving, and visualizing data produced during developers' interactive debugging activities.

SDI is an Eclipse IDE [1] plug-in and a set of servers, which capture Java Platform Debugging Architecture (JPDA) events during a debugging session. When a developer creates a debugging session within Eclipse, the SDI starts two listeners, and it uses RESTful messages to communicate with servers to store debugging data. After that, some tools are available for retrieving and visualizing these data.

SDI is organized into three main modules: (1) Swarm Debug Tracer; (2) Swarm Debug Services; and (3) Swarm Debug Views. Next sections give details about these three modules.

4.1 Swarm Debug Tracer

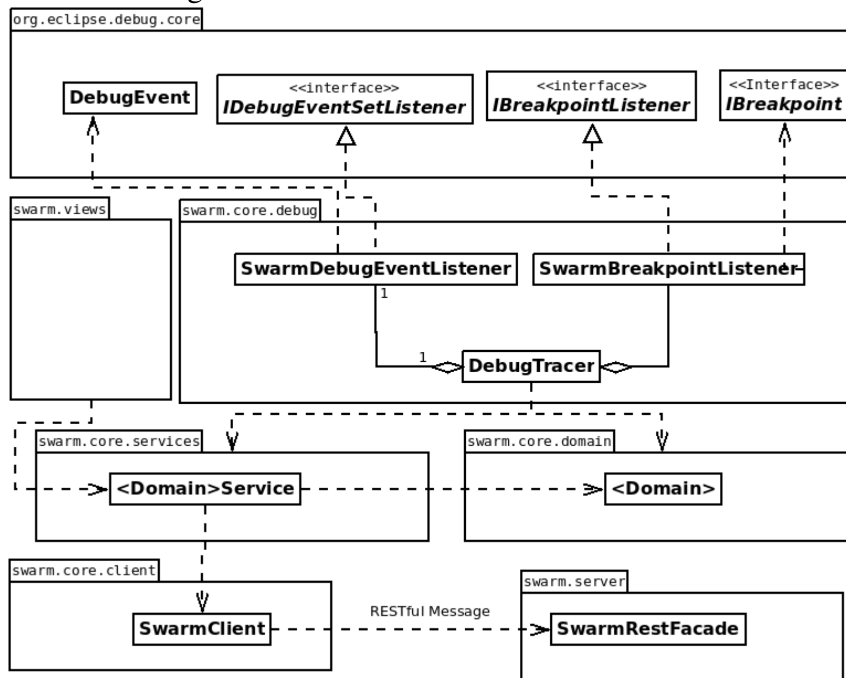
Swarm Debug Tracer (SDT) is an Eclipse IDE plug-in that listens to debugger events during debugging sessions, extending the JPDA. Using the Eclipse JPDA, events are listened by DebugTracer that implements two listeners: `IDebugEventSetListener` and `IBreakpointListener`. Figure 4.1 shows the SDT architecture.

Swarm Debug Tracer has two listeners. First, `SwarmDebugEventListener` implements `org.eclipse.debug.core. IDebugEventSetListener` interface, and it intercepts all user interactions during a Eclipse debug session, as *Step Into (F5)* or *Step Over (F6)*. The second listeners is `SwarmDebugBreakpointListener` that implements `org.eclipse.debug.core. IBreakpointListener`. It intercepts all breakpoints events, but SDT only collects data when a breakpoint is added, and associates it with its Eclipse `IType`.

After an authentication process, the developer creates a Swarm Debug Session using the Swarm Manager view (Figure 4.2). At this point, the developer can toggle breakpoints or start a normal Eclipse debugging session, with stepping events as *Step Into*, *Step Over* or *Step Return*. All these events are captured **silently and transparently**

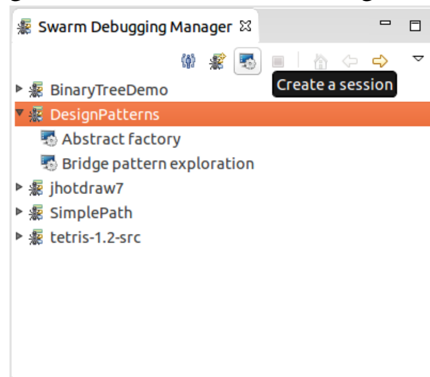
[1] https://www.eclipse.org/

Figure 4.1: The Swarm Tracer architecture



Source: from author

Figure 4.2: The Swarm Manager view



Source: from author

by STD. Next, these events are caught, and stack trace items are analyzed by the Tracer, extracting method invocations only for explicitly visited methods (*Step Into* event).

To use SDT, the developer must open the view “Swarm Manager" and establish a connection with the Swarm Debugging Services. Target projects must be previously associated with the Swarm Manager for being debugged within SDI. This association consists in linking a Swarm Session with a project in the Eclipse workspace. Once a session is established, the developer can use any feature of the regular Eclipse debugger, and the SDT collects the interaction events in the background, with no visible performance decrease.

Typically, the developer will toggle some breakpoints to stop the execution of the program of interest at locations deemed relevant to fix the fault at hand. The SDT collects

the data associated to these breakpoints (locations, conditions, and so on). After toggling breakpoints, the developer runs the program in debug mode. The program stops at the first reached breakpoint. Consequently, for each event, such as *Step Into* or *Breakpoint*, the SDT captures the event and related data. It also stores data about methods called, storing invocations entry for each pair invoking/invoked method. Following the foraging approach (PIORKOWSKI; FLEMING, 2013), the SDT only collects invoking/invoked methods that were visited by the developer during the debugging session, ignoring other invocations. The debugging activity continues until the program execution finishes. The Swarm session is then completed.

To avoid performance and memory issues, the SDT collects and sends the data using a set of specialized *DomainServices* that send RESTful messages to a *SwarmRestFacade*, connecting to the Swarm Debug Services.

4.2 Swarm Debug Services

The Swarm Debug Services (SDS) provide the infrastructure needed by the Swarm Debug Tracer (SDT) to store and, later, share debugging data among developers. Figure 4.3 shows the architecture of this infrastructure. The SDT sends RESTful messages that are received by a SDS instance, which stores them in three specialized persistence mechanisms: a SQL database (PostgreSQL), a full-text search engine (ElasticSearch), and a graph database (Neo4J).

The three persistence mechanisms use similar sets of concepts to define the semantics of the SDT messages, i.e., the concepts specified in the meta-model shown in Chapter 3, Figure 3.3. Figure 4.4 shows these concepts using an entity-relationship representation.

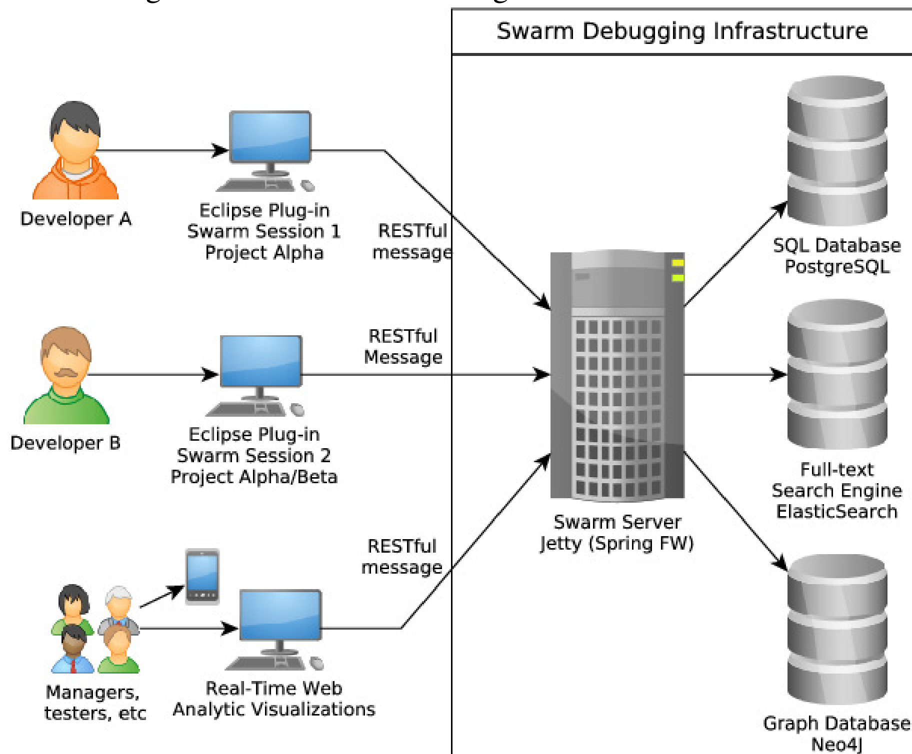
The SDS provide several services for manipulating, querying, and searching collected data: (1) Swarm RESTful API; (2) SQL query console; (3) full-text search API; (4) dashboard service; and (5) graph querying console.

4.2.1 Swarm RESTful API

The RESTful API allows manipulating debugging data using the Spring Boot framework[2]. Create, retrieve, update, and delete operations are available through HTTP GET requests and respond with a JSON structure. For example, upon submitting the request:

[2]http://projects.spring.io/spring-boot/

Figure 4.3: The Swarm Debug Services - architecture



Source: from author

```
http://swarmdebugging.org/developers/search/findByName?
name=petrillo
```

the SDS responds with a list of developers whose names are “petrillo", in JSON format.

4.2.2 SQL Console

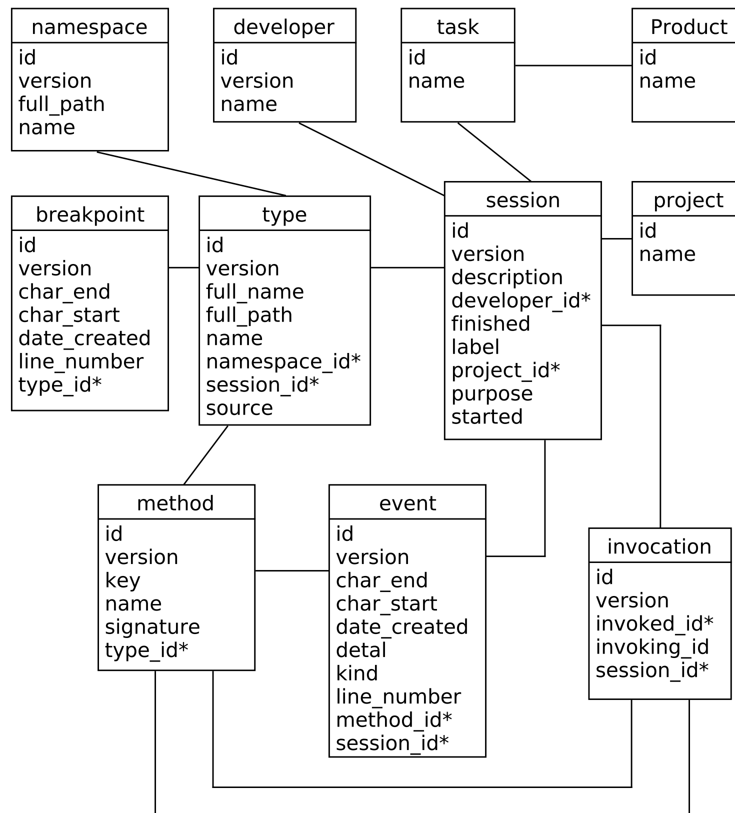
The SDS provides a SQL console available at <http://db.swarmdebugging.org> to receive queries on debugging data, providing relational aggregations and functions. Figure 4.5 shows an example of SQL query on breakpoint data.

4.2.3 Full-text Search Engine

The SDS also provides an ElasticSearch[3] feature, which is a highly scalable open-source full-text search and analytic engine, to store, search, and analyze the debugging data. The SDS creates an instance of the ElasticSearch engine and offers a console for executing complex queries on the debugging data.

[3]https://www.elastic.co/

Figure 4.4: The Swarm Debug metadata



Source: from author

Figure 4.5: Swarm Debug SQL Console

PostgreSQL 9.3.12 rodando em **localhost**:5432 -- Você está logado como usuário "**swarm**" SQL | Histórico | Encontrar | Sair

phpPgAdmin: PostgreSQL: swarm:

Esquemas SQL Encontrar Variáveis Processos Travas Administração Privilégios Exportar

Informe o SQL a executar abaixo:

SQL

```
SELECT * FROM breakpoint WHERE type = 200
```

ou carregue o script SQL de um arquivo: Choose file No file chosen

Paginar resultados

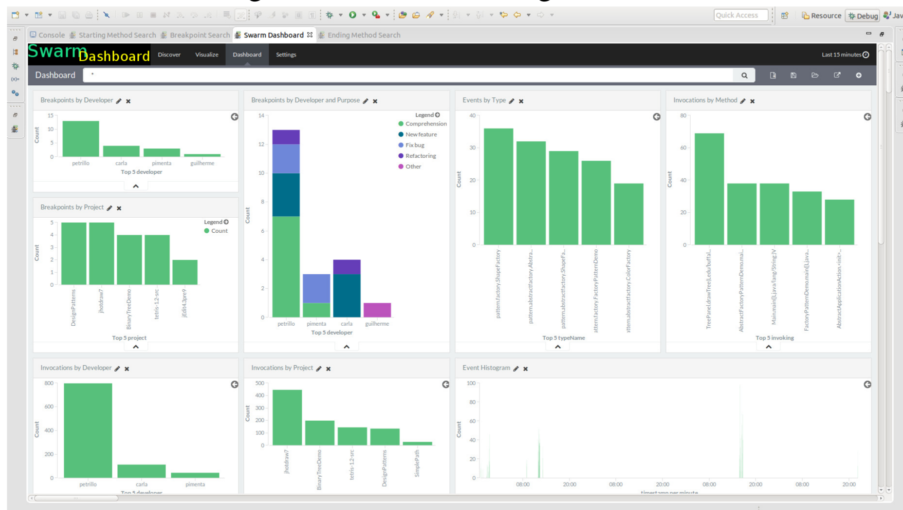
Executar Reiniciar

Source: from author

4.2.4 Dashboard Service

The ElasticSearch allows the use of the Kibana dashboard, and we expose a Kibana instance on Swarm Debugging collect data. With this dashboard, researchers can build charts for representing the retrieved data. Figure 4.6 shows a Swarm Dashboard embedded into Eclipse as a view.

Figure 4.6: Swarm Debug Dashboard



Source: from author

4.2.5 Graph querying console

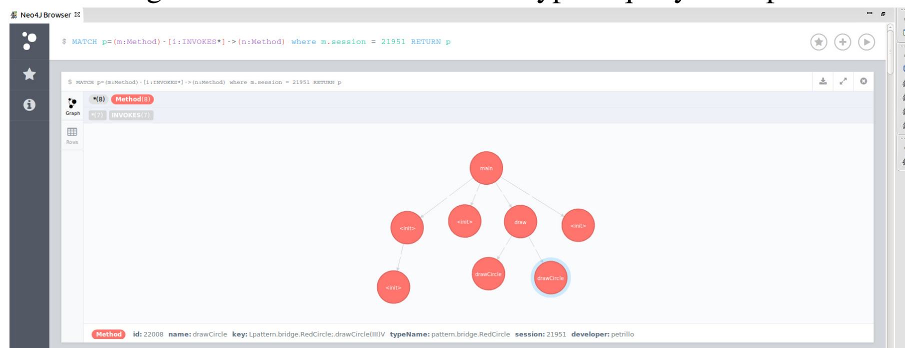
The SDS also persists debugging data in a Neo4J[4] graph database. Neo4J provides a query language named Cypher, which is a declarative, SQL-inspired language for describing patterns in graphs. It allows researchers to express what they want to select, insert, update, or delete from a graph database without describing precisely how to do it. Thus, we expose a Neo4J Browser and creates an Eclipse view. Figure 4.7 shows an example of a Cypher query and the resulting graph.

4.3 Swarm Debug Views

On top of the SDS, the SDI provides several tools for searching and visualizing the data collected during debugging sessions. These tools are integrated into the Eclipse IDE, simplifying their usage. All graph views were implemented using CytoscapeJS (SAITO et al.,), a JavaScript Graph API framework. As a web application, the SD visualizations

[4]http://neo4j.com/

Figure 4.7: Neo4J Browser - a Cypher query example



Source: from author

can be integrated into an Eclipse view as a SWT Browser Widget, or accessed through a traditional browser such as Mozilla Firefox or Google Chrome. It can also be accessed through tablets and smartphones from a web browser. In the next sections, we present these views.

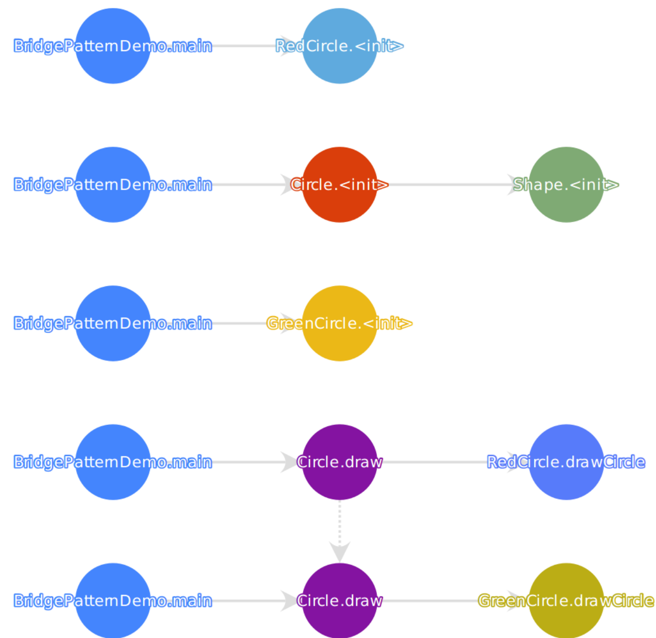
4.3.1 Sequence stack diagram

The sequence stack diagram, shown in Figure 4.8, is a novel diagram (PETRILLO et al., 2015) that represents sequences of method invocations. Circles represent methods and arrows represent invocations, and each line is a complete stack trace, without returns. The first node is a starting method (non-invoked method), and the last node is an ending method (non-invoking method). If an invocation chain contains a non-starting method, a new line is created, and the current stack is repeated, and a dotted arrow is used to represent a return to this node, as illustrated by the method *Circle.draw* in Figure 4.8. Furthermore, developers can directly go to a method in the Eclipse Editor by double-clicking on a node in the diagram.

4.3.2 Dynamic method call graphs

A dynamic method call graph is a direct call graph (GROVE et al., 1997), shown in Figure 4.9, used to display the hierarchical relations between invoked methods. Again, circles are used to represent methods and oriented arrows to express invocations. Each session generates a graph, and all invocations collected during a session are shown in the corresponding graph. The circles corresponding to starting points (non-invoked methods) are drawn on top of a tree, and adjacent nodes represent invocations sequences. Developers can navigate sequences of methods invocations pressing the F9 (forward) and F10

Figure 4.8: Sequence stack diagram for Bridge design pattern



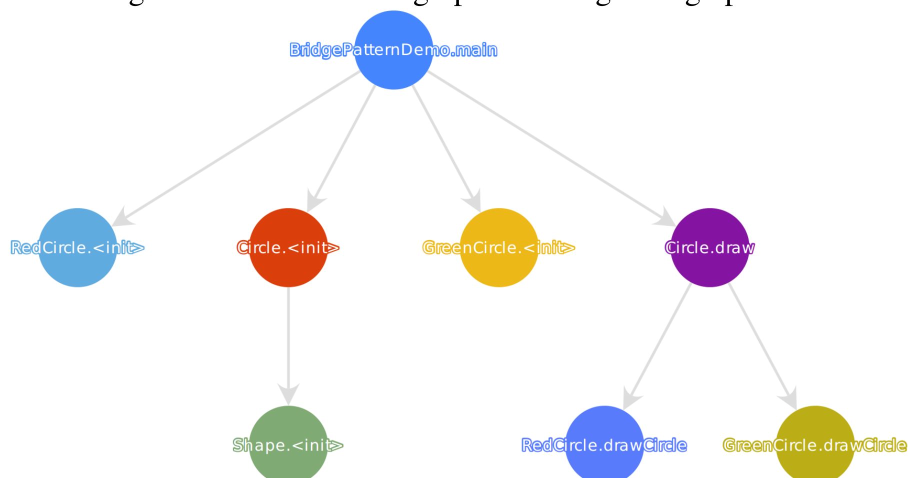
Source: from author

(backward) keys. They can also go directly to a method in the Eclipse Editor by double-clicking on nodes in the graph.

4.3.3 Debug Global View

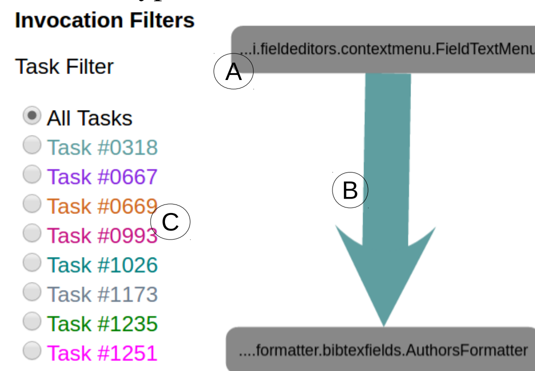
Debug Global View (GV) is a call graph for modeling software based on directed call graphs (GROVE et al., 1997) to represent the hierarchical relationships created by methods invocations. This visualization use rounded gray boxes (Figure 4.10-A) to represent types or classes (nodes) and oriented arrows (Figure 4.10-B) to express invocations

Figure 4.9: Method call graph for Bridge design pattern



Source: from author

Figure 4.10: GV elements. A: types (nodes); B: invocations (edges); C: task filter area.



Source: from author

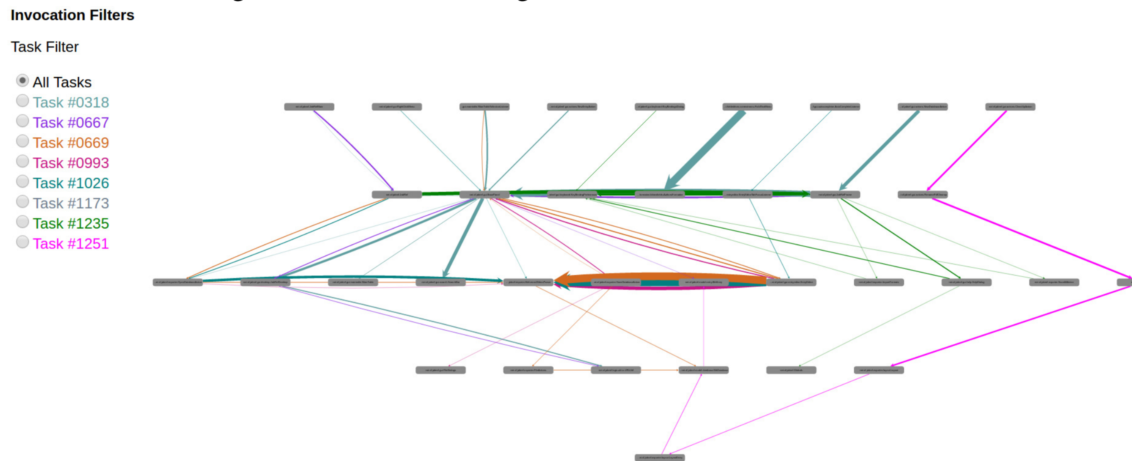
(edges). GV is built using context data collected in previous debugging sessions while developers performed different tasks.

In GV, the gray boxes are types that developers visited during debugging sessions. The edges represent method calls (*Step Into* or F5 on Eclipse) performed by all developers in all traced tasks in a software project. Each edge color describes a task and the line thickness is proportional to the number of invocations, i.e., the number of times a developer followed that path. Each debugging session contributes with a context, generating the visualisation combining all collected invocations. The visualization is organized in layers or stacks, and each line is a layer defined by the invocations, applying an automatic *breadthfirst* layout manager. The starting points (non-invoked methods) are allocated on top of the tree, the adjacent nodes following the invocations sequence. Also, developers can directly go to a type in the Eclipse Editor by double-clicking on a node in the diagram. On the left side of this view, developers find radio buttons that can be used to filter invocations by task (Figure 4.10-C), showing the paths used by developers during previous debugging sessions. Finally, developers can use the mouse to pan and zoom in/out on the view. Figure 4.11 shows an example of GV with all tasks performed in debugging the JabRef system, and from which we have data about seven tasks [5].

GV is a contextual visualization that uses only paths explored by developers. Thus, GV shows only the paths explicitly and intentionally visited by developers, including types declarations and methods invocations explored by developers based on their decisions.

[5]Task 1173 is empty in this dataset.

Figure 4.11: GV showing data from all tasks with JabRef.



Source: from author

4.3.4 Breakpoint search tool

The breakpoint search tool can be used by researchers and developers to find suitable breakpoints when working with the debugger. For each breakpoint, the SDS captures the type and location of the type where the breakpoint was toggled. Thus, developers can share their breakpoints. The breakpoint search tool allows *fuzzy*, *match*, and *wildcard* ElasticSearch queries. Results are displayed in the Search View table for easy selection. Developers can also open a type directly in the Eclipse Editor by double-clicking on a selected breakpoint. Figure 4.12 shows an example of breakpoint search, in which the search box contains the misspelled word *fcatory*.

Figure 4.12: Breakpoint search tool (fuzzy search example)

Console | Swarm Debugging Manager | Starting Method Search | Breakpoint Search | Ending Method Search

Project: DesignPatterns   Search: fcatory

Developer	Label	Purpose	Type	Descriptio
guilherme	Abstract factory analysis	Other	pattern.abstractfactory.Abs	To navegal
petrillo	Abstract factory	Fix bug	pattern.abstractfactory.Abs	
carla	Factory Pattern	Refactoring	pattern.factory.FactoryPatte	Try refactc

Source: from author

4.3.5 Starting/Ending method search tool

The starting/ending method search tool allows searching for methods that (1) only invoke other methods but are not explicitly invoked themselves during the debugging session, and (2) are only invoked by other methods but do not invoke others.

Formally, we define Starting/Ending methods as follows. Given a graph $G =$

(V, E), where V is a set of vertexes $V = \{V_1, V_2, \ldots, V_n\}$ and E is a set of edges $E = \{(V_1, V_2), (V1, V3), \ldots\}$. Then, each edge is formed by a pair: $< V_i, V_j >$, were V_i is the *invoking* method and V_j is the *invoked* method. If α is the subset of all vertices *invoking* methods and β is the subset of all vertices *invoked* by methods, then the Starting and Ending methods are:

$$StartingPoint = \{V_{SP} \mid V_{SP} \ \in \alpha \; and \; V_{SP} \notin \beta\}$$

$$EndingPoint = \{V_{EP} \mid V_{EP} \ \in \beta \; and \; V_{EP} \notin \alpha\}$$

Locating these methods is important in a debugging session because they are the entry and exit points of a program at runtime.

4.4 Definition of Co-Breakpoint

The concept of coupling has been defined for interaction and change (ZOU; GODFREY; HASSAN,): interaction coupling and change coupling (also named co-change). Interaction traces are used to detect interaction coupling and predict interaction (ZOU; GODFREY; HASSAN,). Change histories are used to detect change coupling (*i.e.,* co-change) and predict change (ZIMMERMANN et al., 2005). Recently, Bantelay *et al.* combined both interaction coupling and co-change to improve interaction and change prediction (BANTELAY; ZANJANI; KAGDI, 2013).

Similar to the concepts of interaction coupling and co-change, in SD we introduce the concept of co-breakpoint. Let X and Y be two program entities: X and Y are co-breakpoints if a developer that toggles the breakpoint in entity X, also toggles the breakpoint in entity Y.

Let $S = \{e_1, e_2, ..., e_n\}$, be a set of entities involved in the debugging session. A co-breakpoint is the association rule defined by $X \Rightarrow Y$ where X and Y are a pair or distinct entities in S in which a developer toggles breakpoints.

In SD, we use debugging activities collected by SDI to mine co-breakpoints. We aim at using co-breakpoints to predict breakpoint, *i.e.,*, to recommend a program entity where a developer may need to toggle a breakpoint to locate the fault.

Table 4.1: Illustration of co-breakpoint

Tasks	Breakpoint Activities	co-breakpoint(Task)
T_1	$e_1 \rightarrow e_2 \rightarrow e_3$	$\{e_1, e_2, e_3\}$
T_2	$e_4 \rightarrow e_5 \rightarrow e_2 \rightarrow e_6$	$\{e_4, e_5, e_2, e_6\}$
T_3	$e_1 \rightarrow e_7$	$\{e_1, e_7\}$
T_4	$e_2 \rightarrow e_5 \rightarrow e_3$	$\{e_2, e_5, e_3\}$

4.5 Breakpoint Prediction

This section presents the approach that we use to predict breakpoints. We follow a recommendation process similar to Lee *et al.* (LEE et al., 2015): we mine the co-breakpoints on the fly where the co-breakpoints for each task are used as a test set and training set. To illustrate the prediction process, consider Table 4.1, where developers have a set of tasks $T = \{T_1, T_2, T_3, T_4\}$ to perform. We define the function $co - breakpoint(T)$ that returns the ordered set of the entities in which the developer has toggled breakpoints when resolving the task T. The first task T_1 shows the co-breakpoint between the entities $\{e_1, e_2, e_3\}$ (*i.e.,* $co - breakpoint(T_1) = \{e_1, e_2, e_3\}$). Similarly, $co - breakpoint(T_2) = \{e_4, e_5, e_2, e_6\}$, $co - breakpoint(T_3) = \{e_1, e_7\}$, and $co - breakpoint(T_4) = \{e_2, e_5, e_3\}$.

Mining the co-breakpoints on the fly consists of ordering the tasks by their completion date (*i.e.,*, to predict the breakpoint for a task using the co-breakpoint mined from an earlier task). The breakpoint prediction is illustrated in Table 4.2. Each line in Table 4.2 shows the current task, the content of the co-breakpoint database (DB), and the recommendation candidates for each breakpoint in the co-breakpoint set. For example, the first line shows that when performing T_1 (T_1 is used for the testing set), the co-breakpoint database is empty, and for each breakpoint in $co-breakpoint(T_1)$, there is no recommendation candidate (as the database is empty). For the second line, T_1 is used as a training set. The database contains the co-breakpoint of T_1 (which is already used as the testing set). For the current breakpoint in $co-breakpoint(T_2)$ (*e.g.,* e_4), we look into the database and found that there is no breakpoint in the database that shares the co-breakpoint relation with e_4. Thus, the recommendation candidates are empty for e_4, and the same for e_5. On the contrary, the database shows that e_2 has e_1 and e_3 as co-breakpoints. So they are candidates for recommendation. At the end of T_2, the co-breakpoint set is added to the database (see the line for task T_3). The process repeats for T_3. At the end (T_4), we can see that when a developer toggles a breakpoint on e_5, the candidates for recommendation do not include e_2 because a breakpoint was already toggled in it.

Table 4.2: Approach for breakpoint prediction

Tasks	Co-breakpoint Database (DB)	Recommendation Candidates
T_1	$\{\}$	$(e_1, \varnothing) - (e_2, \varnothing) - (e_3, \varnothing)$
T_2	$\{\{e_1, e_2, e_3\}\}$	$(e_4, \varnothing) - (e_5, \varnothing) - (e_2, \{e_1, e_3\}) - (e_6, \varnothing)$
T_3	$\{\{e_1, e_2, e_3\}, \{e_4, e_5, e_2, e_6\}\}$	$(e_1, \{e_2, e_3\}) - (e_7, \varnothing)$
T_4	$\{\{e_1, e_2, e_3\}, \{e_4, e_5, e_2, e_6\}, \{e_1, e_7\}\}$	$(e_2, \{e_1, e_3, e_4, e_5, e_6\}) - (e_5, \{e_4, e_6\}) - (e_3, \{e_1\})$

At each stage of the breakpoint prediction, we compute precision and recall metrics, and use them to evaluate the accuracy of the prediction. To compute them, we use the set of recommended candidates (C) and the set of co-breakpoint (B) for the given task.

$$Precision = \frac{|B \cap C|}{|C|} \quad Recall = \frac{|B \cap C|}{|B|} \tag{4.1}$$

4.6 Use scenarios

Swarm Debugging provides several practical use scenarios, and to exemplify, four possible scenarios are described in the following subsections.

Scenario A - finding suitable breakpoints: finding suitable breakpoints is the first challenge addressed by Swarm Debugging. First, using the **Breakpoint search tool**, developers can find breakpoints toggled by other programmers in the same project. Searching for a label, description, purpose, developer or full-text of the breakpoint line of code, this tool provides a shared memory of relevant points to toggle a breakpoint. Second, if programmers don't find a breakpoint to reach their object, they can use the **Starting point search tool**. Starting points are the initial methods in a runtime session, and they are recurrent points to start a code exploration (LATOZA; MYERS, 2010). In addition, the source code full-text search tool is a generic tool to find breakpoint candidates. Finally, visualizing diagrams of previous sessions, developers can have useful insights for toggling their next breakpoints.

Scenario B - visualizing debugging sessions: Swarm Debugging can be used to visualize steps in real-time during a debugging session. Using the *Sequence Stack Diagram* or the *Method Call Graph*, developers have a full memory of their steps, and they can visualize just-in-time the sequence and relationships between methods. Also, these session visualizations can be revisited for new tasks. Finally, complicated debugging executions can be divided into multiples Swarm sessions, simplifying a complex problem in various simple views.

Scenario C - creating collective intelligence: since Swarm debugging is based on

collecting data, it is useful for sharing knowledge about a software project. For instance, developers can know which project areas have been explored before and by whom, or why a breakpoint was created by a colleague. This information is likely to improve and create new interactions among members of a software team.

Scenario D - Managing projects: Swarm Debugging opens new perspectives for managing software projects. Using web technologies, project managers can use the visualizations and *Swarm Dashboard* to create real-time panels, visualizing collected information, which is lost in other approaches. For example, managers can query the most visited areas, or study developers' debugging patterns or breakpoint patterns. Additionally, Swarm Debugging databases provide new opportunities for empirical studies about debugging and software quality.

4.7 Final remarks

Swarm Debug Infrastructure provides an open-source infrastructure, integrated into Eclipse, to collect, store and share interactive debugging session data, contextualizing breakpoints and events during these sessions. Further, it provides real-time and interactive visualizations using web technologies, making an automatic memory of developer exploration paths. Moreover, dividing software exploration by sessions and representing them by call graphs allows easy understanding because only intentional visited areas are shown on these graphs. One can through the execution of a project and see only the important areas that are relevant to developers. SDI stores data on a remote server using an asynchronous execution and, thus, does not suffer from performance or memory issues like omniscient debuggers (POTHIER; TANTER, 2009) or tracing-based approaches (OHMANN; LIBLIT, 2016).

Currently, the Swarm Tracer is implemented in Java, using Eclipse Debug Core services. However, SDI provides a RESTful API that can be accessed independently, and new tracers can be implemented for different IDEs or debuggers. The research community can leverage SDI by conducting more studies to improve our understanding of developers' debugging activities, which could ultimately result in the development of whole new families of debugging tools that are more efficient and–or more adapted to the particularity of each debugging activity.

In conclusion, to the best of our knowledge, others tools like Hipikat, Jive or DebugAdvisor while tools for debugging support, they do not consider to explore collective

aspects, as SD implemented on SDI. Thus, the SDI complements previous approaches by collecting data that allow researchers to study how developers find investigate project during debugging sessions, possibly restoring parts of the developers' contexts after interruptions by recalling previous breakpoints or stepping invocations, for example.

5 EVALUATION OF THE SWARM DEBUGGING

In order to assess SD and to evaluate the effectiveness of SDI, we performed three studies using SDI. First, we conducted an experiment with three actual maintenance task performed by seven developers on JabRef system. We aimed to evaluate how useful the data collected by SDI could be use to understand interactive debugging. Next, we conducted a study collecting debugging activities of 20 developers when performing realistic maintenance tasks on a Java open-source system, focusing on how developers toggle breakpoints. Finally, we assessed the usefulness of GV for conducting a qualitative study with 23 professional developers and a controlled experiment with 13 professional developers. The tree studies are described as follows.

5.1 Experiment 1 - towards understanding interactive debugging

In this section, we present the study on the use of SDI to collect and share debugging activities. This study aims to evaluate how the data collected by SDI could be useful. Thus, we use the data collected by SDI to answer 5research questions. We first present the context of the study. Then, we explain the design and report the results of the study.

We assess the effectiveness of the Swarm Debugging and SDI through a first experiment that aims to understand how developers apply interactive debugging on five actual faults found in JabRef, toggling breakpoints, and stepping code. Our study involved five freelancers and two student developers performing 19 bug location sessions. We collect videos recording and data about 6 hours of effective debugging activities. The data includes 110 breakpoints and near 7,000 invocations. We process the collected videos and data to answer 5research questions showing that (1) there is no correlation between the number of invocations (respectively the number of breakpoints toggled) during a debugging session and the time spent on the debugging task; $\rho = -0.039$ (respectively 0.093). (2) there is no correlation between numbers of breakpoints and elapse task time ($\rho = 0.093$); We also observed that (2) developers follow different debugging patterns and (3) there is no relation between numbers of breakpoints and expertise. However, (4) there is a strong negative correlation between time of the first breakpoint ($\rho = -0.637$); and the time spent on the task, suggesting that when developers toggle breakpoints carefully, they complete tasks faster than developers who toggle breakpoints too quickly.

5.1.1 Context

Studies and discussions about interactive debugging are scarce in the literature of program comprehension, so we could elaborate many research questions to better understand such important software development activity. However, to illustrate the use of the SDI, we formulate the following 5 specific research questions:

RQ1: Is there a correlation between the numbers of invocations and tasks' elapsed time?

RQ2: Is there a relationship between the number of breakpoints and tasks' elapsed time?

RQ3: Do developers explore/debug in different ways a task?

RQ4: Is there a correlation between the numbers of breakpoints and developers' expertise?

RQ5: Is there an association between time of first breakpoint and task's elapsed time?

To answer the research questions above, we run the experiment designed as described in the next section.

5.1.2 Study Design

We have planned the study as follows.[1] We had to choose debugging tasks to trigger participants' debugging activities. We chose to ask participants to find the locations of true faults in an independent, open-source program. We selected JabRef[2] as target program, which is an open-source bibliography reference manager developed in Java. We chose JabRef because it has faults publicly reported in its issue tracker and its domain was easy to understand by the participants. We picked five faults reported against JabRef v3.2 in its issue tracker and asked participants to find the locations of the faults described in issues 318, 667, 669, 993 and 1026.

In order to estimate task's effort, we calculated averages of elapsing time for each task by a participant. Table 5.1 shows the average time (in minutes) for each task and the global average. In average, participants spent **21 minutes** to complete the bug location tasks.

To reproduce a realistic industry scenario, we recruited five professional freelancer developers[3]. Among them, 2 Java experts and three intermediates, 100% were male,

[1] All artifacts on <http://swarmdebugging.org/publications>.
[2] <http://www.jabref.org/>
[3] https://www.freelancer.com/

Table 5.1: Elapse time by task (average)

Task	Time (min.)
318	13
667	31
669	11
993	28
1026	21
Mean	**21**

100% used Eclipse and 100% used debuggers frequently. As many other experimental studies, we asked two volunteer students at Polytechnique Montréal to participate in our experimental study.

We provided participants with instructions by two documents. The first document was an experiment tutorial[4] which explained how to install and configure all tools to perform a warm-up task and the experimental study. We also used the warm-up task to confirm that the participants' environments were correctly configured and that the participants understood the instructions. The warm-up task was described using a video to guide the participants. We make available this video on-line[5].

The second document presented the five issues with a description and some piece of information to reproduce the faults. To reduce the participants' effort to reproduce the faults, we offered videos demonstrating step-by-step how to reproduce the faults. We also provided the participants with an electronic form to report whether they were tired or not at the end of the experiment.

For this experimental study, we used Eclipse Mars 2 and Java 8, the SDI and its Swarm Debug Tracer plug-in, and two Java projects: a small Tetris game for the warm-up task and JabRef v3.2 for the experimental study. All participants received the same workspace, provided by our artifact repository.

After installing the environment (Eclipse and the SDI), each participant executed the warm-up task. This tasks consisted of starting a debugging session, toggling a breakpoint, and debugging a Tetris program to locate a given method. After the warm-up task, each participant executed debugging sessions to find the location of faults described in the five selected issues. We did not set a time constraint but suggested 20 minutes per fault. We asked participants to control their fatigue, asking them to go to the next task if they felt tired while informing us of this situation in their reports. Finally, each participant filled a report to provide their answers and other information, whether they completed the

[4]<http://swarmdebugging.org/publications/experiment/tutorial.html>
[5]<https://youtu.be/U1sBMpfL2jc>

tasks successfully or not.

All services were available on our server[6] during these debugging sessions and the experimental data were collected in the course of 8 days. We also capture video images of the sessions. The experiment tutorial contained the instruction to install and set the OBS (Open Broadcaster Software), an open source system for live streaming and recording[7]. Participants were asked to provide the video captured during the experiment. A video was recorded for each task, providing about **6 hours** of effective developer's activities. We had 19 completed tasks by five developers, 110 collected breakpoints, and more than 6000 invocations.

After the participants had completed the debugging sessions (successfully or not), we used the tools provided by the SDI on the data collected to answer each research question. To answer RQ1 and RQ2, we used SQL queries, with which we can extract all the invocations and breakpoints set during each session and find a relationship between breakpoints and tasks. The example of SQL to extract data to RQ2 is:

```
select t.id taskId, s.id sessionId, count(*) from breakpoint b, task t,
type tp, session s where b.type_id = tp.id and tp.session_id = s.id
and s.task_id = t.id group by s.id, t.id order by s.id
```

Finally, to answer RQ3, we plotted the call graph of each debugging session using the SDI. We organized these graphs by tasks and by numbers of invocations, analyzing each graph to identify navigation patterns. The SQL to extract data to RQ3 is:

```
select s.developer_id, tsk.title, s.id, count(*) as invocations
from  product p, task tsk, session s, invocation i
where p.id = 1 and p.id = tsk.product_id and tsk.id = s.task_id
and i.session_id = s.id group by tsk.title, s.id
order by tsk.title,invocations
```

Next, we report the results of our analyses to answer our research questions.

5.1.3 RQ1: Is there a correlation between the numbers of invocations and tasks' elapsed time?

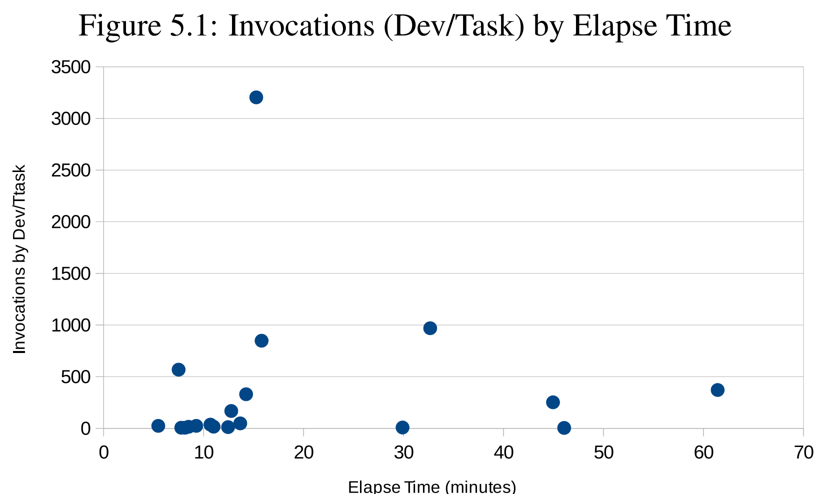
We define the debugging patterns based on when developers toggle breaking during the maintenance task.

[6]<http://server.swarmdebugging.org>
[7]https://obsproject.com

- We define the concept of "breakpoint in the first half" and breakpoint in the second half. A breakpoint is in the first half if it is toggled before the 50% of the time spend to perform the maintenance task. Otherwise, the breakpoint is in the second half.
- We compute the percentage of the breakpoint in the first and the second half. Plot the percentage and see if we can have a cutoff point to split the data. If yes, we can have a different pattern using the cutoff point to distinguish different breakpoint patterns: **breakpoint first pattern and breakpoint last pattern**.
- Using the two patterns, we can relate them with the developer's profile i.e., follow a breakpoint pattern is related to developers' profile?
- RQ3: Relate the patterns to the accuracy and relate the pattern to the overall time spent to perform the task

By analyzing the elapse time of each task executed by developer and invocations, we can plot Figure 5.1 in which we can observe that **there is not a correlation between the numbers of invocations and elapse task time**. This conclusion can be strengthened by the Pearson's correlation coefficient ($\rho = -0.039$) lower than 0.1.

Figure 5.1: Invocations (Dev/Task) by Elapse Time

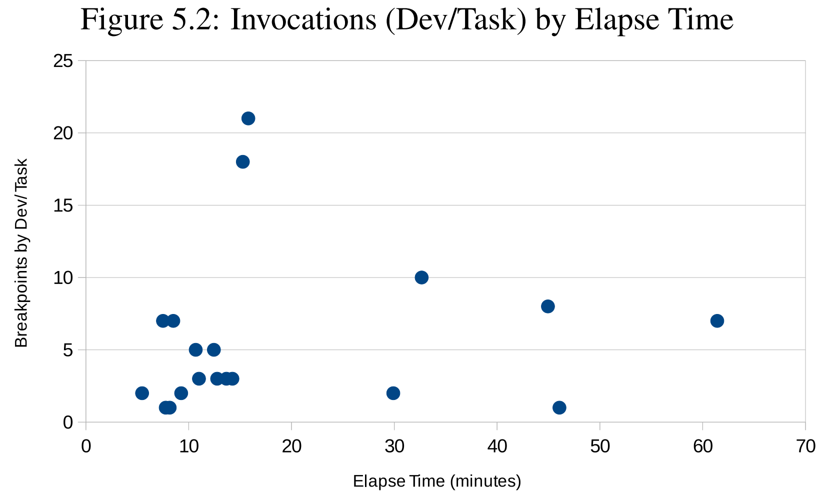


Source: from author

5.1.4 RQ2: Is there a relationship between the number of breakpoints and tasks' elapsed time?

By analyzing the elapse time of each task executed by developer and breakpoints, we can plot Figure 5.2 in which we can observe that **there is not a correlation between the numbers of toggled breakpoints and elapse task time**. This conclusion can be strengthened by the Pearson's correlation coefficient ($\rho = 0.093$) lower than 0.1.

Figure 5.2: Invocations (Dev/Task) by Elapse Time



Source: from author

5.1.5 RQ3: Do developers explore/debug in different ways a task?

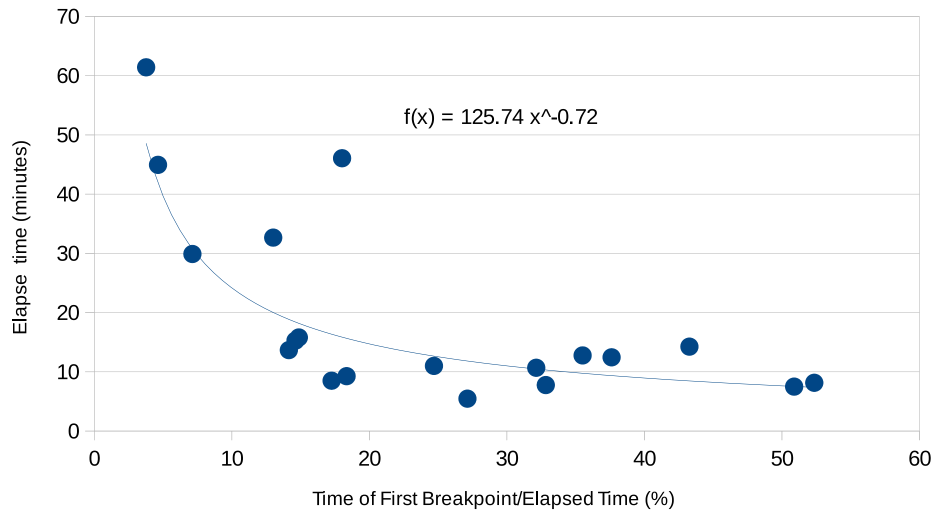
We observed two distinct debugging navigation patterns: (1) a *fuzzy* debugging pattern and (2) a *straight* debugging pattern. In the fuzzy debugging pattern, the call graph presents several branches, showing that participants used a foraging approach. Figures 5.4 and 5.5 show two typical fuzzy debugging graphs. In the straight debugging pattern, participants followed a straight or quasi-straight set of invocations, as shown in Figure 5.6 and 5.7.

Furthermore, we identified a strong correlation between expertise and navigation patterns: the more expert the participants, the more straightforward their navigation patterns. Future work will further study this correlation to confirm its existence and provide explanations and, possibly, recommendations to developers during debugging activities.

5.1.6 RQ4: Is there a correlation between the numbers of breakpoints and developers' expertise?

By relating the numbers of breakpoints toggled during debugging tasks and developers' expertise, we can conclude that **there is no relation between numbers of breakpoints and expertise**. Although this result may seem counter-intuitive, because of the more expert participants, the fewer breakpoints she could need, we explain this result on three possible explications. First, the numbers of breakpoints are possibly more related to task complexity. Second, all participants were newcomers to JabRef. Third, the chosen program and issues are not representative of all programs and debugging tasks.

Figure 5.3: Relation between time of first breakpoint and task elapsed time



Source: from author

5.1.7 RQ5: Is there an association between time of first breakpoint and task's elapsed time?

Breakpoints are key for interactive debugging, and an important breakpoint is a first toggle breakpoint during a session. We analyzed 19 interactive debugging sessions in which 73% (14/19 sessions) started a first debugger execution after lower than 3 minutes of toggled first breakpoint, and 52% (10/19 sessions) started the first debugger immediately (lower than 10 seconds) after had defined the first breakpoint. In conclusion, a first breakpoint is an important decision on an interactive debugging session.

To analyze if there is a relation between time of the first breakpoint and task elapsed time, for each session, we normalized our data dividing each first breakpoint time by task elapsed time, and we associated this ratio with its respective elapsed time, plotting Figure 5.3.

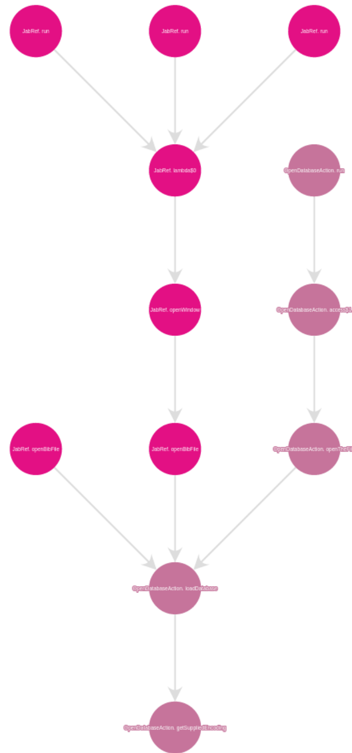
Analysing Figure Figure 5.3, it is clear that **there is a strong correlation between time of first breakpoint** ($\rho = -0.637$), and task elapsed time is **inversely proportional** to the time of task's first breakpoint, following a correlation function:

$$f(x) = \frac{\alpha}{x^{\beta}} \tag{5.1}$$

where $\alpha = 125$ and $\beta = 0.72$.

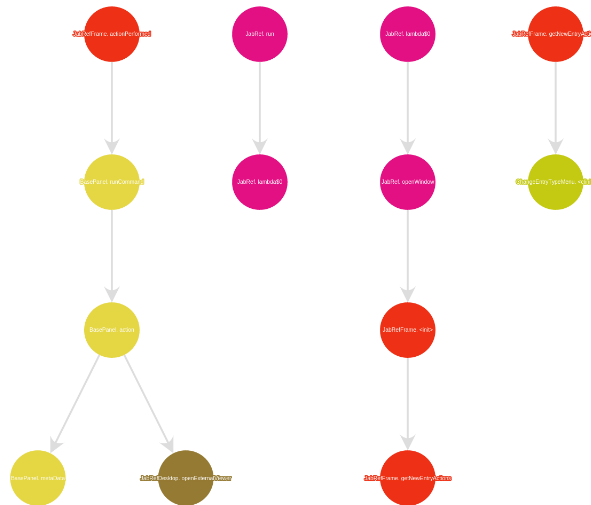
On the whole, results show that **whether developers toggle breakpoints carefully, they complete tasks faster than developers who toggle breakpoints quickly**.

Figure 5.4: Examples of fuzzy debugging patterns - Task 1



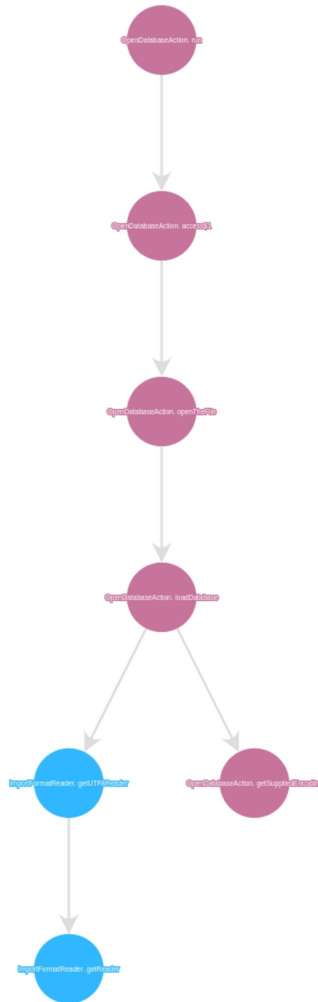
Source: from author

Figure 5.5: Examples of fuzzy debugging patterns - Task 2



Source: from author

Figure 5.6: Examples of straight debugging pattern - Task 1



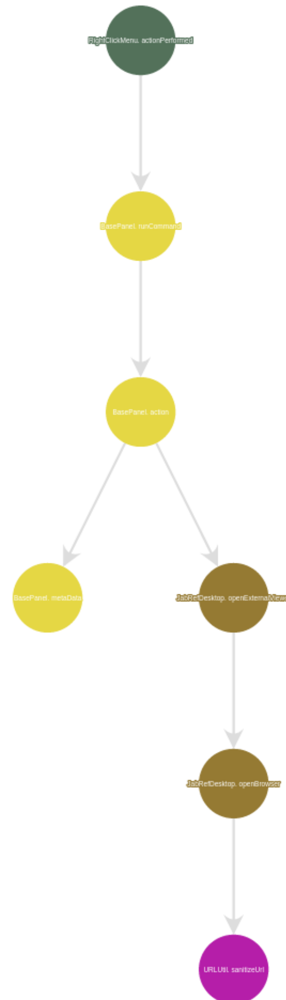
Source: from author

5.1.8 Threats of Validity

As any empirical study, this experimental study is subject to limitations that threaten the validity of its results. The first limitation concerns to the number of participants involved in the study. With 7 participants, we can not claim generalization of the results. However, we accept this limitation because the goal of the study was to show the effectiveness of the data collected by the SDI to obtain insights about developers' debugging activities. Future studies with more participants and more systems and even more tasks are needed to confirm or infirm the results of this study.

Other threats to the validity of our results concern their internal, external, and conclusion validity. We accept these threats because the aim of the experimental study was to show the effectiveness of the SDI to collect and share data about developers' interactive debugging activities, not to answer with strong statistical significance the research ques-

Figure 5.7: Examples of straight debugging pattern - Task 2



Source: from author

tions. Future work is needed to perform in-depth experimental studies with these research questions and other, possibly drawn from the questions that developers asked found by Sillito *et al.* (SILLITO; MURPHY; De Volder, 2008).

5.2 Experiment 2 - mining debugging data to recommend breakpoints: an empirical study

In the previous section, we showed that **developers toggle breakpoints carefully, they complete tasks faster than developers who toggle breakpoints quickly**. Consequently, if good breakpoints improve maintenance task, could developers gather and share their breakpoints? How useful is sharing debugging activities for software maintenance? In this section, we address that question, presenting the results from an empirical study about sharing debugging activities. We developed an infrastructure to collect and ana-

lyze data during debugging activities (Swarm Debugging Infrastructure (SDI)), and we conducted an experiment to understand where developers toggle breakpoints on five true faults found in the JabRef Java open source project.

Our study involved 20 developers (12 students and eight professional freelancers), and we collect more than 6 hours of effective developer's activities and 207 breakpoints. Using our data provided by the SDI and video analyses, we answer 5research questions on the nature of breakpoints, showing that breakpoints are strategical locus that are used several times by several developers for different tasks, and we show that previous breakpoints could be used to improve maintenance tasks to future issues. To demonstrate how breakpoints can be useful, we introduce the concept of co-breakpoint. Finally, we discuss some implications of our results for tool developers and future debuggers.

Thus, in this study, we perform an empirical study about the nature of breakpoints. Next, we adopted the concept of co-breakpoint (introduced in section 4.4) and propose an approach to evaluate it for breakpoint prediction.

5.2.1 Experiment setup

This section presents the design of our experiment, which aims to address the following fourresearch questions:

RQ1: How much time do developers spend to toggle the first breakpoint?

RQ2: On what kind of statement do developers toggle their breakpoints?

RQ3: Do developers toggle breakpoints in the same place?

RQ4: How effective is co-breakpoint for breakpoint prediction?

To answer the research questions, we proceeded as follows[8]:

We had to choose debugging tasks to trigger participants' debugging activities. We chose to ask participants to find the locations of true faults in an independent, open-source program. We selected JabRef[9] as target program, which is an open-source bibliography reference manager developed in Java. We chose JabRef because it has faults publicly reported in its issue tracker, its domain was easy to understand by the participants, and it has modules with some relative independent regions. We picked five faults reported against JabRef v3.2 in its issue tracker and asked participants to find the locations of the faults described in issues 318, 667, 669, 993 and 1026.

[8] Artifacts available at <http://swarmdebugging.org/publications>.
[9] <http://www.jabref.org/>

In order to estimate task's effort, we calculated averages of elapsing time for each task per participant. Table 5.1 shows the average time (in minutes) for each task. Furthermore, in average, participants spent **21 minutes** to complete the bug location tasks.

To reproduce a realistic industry scenario, we recruited eight professional freelancer developers[10]. Among them, 2 Java experts and three intermediates, 100% were male, 100% used Eclipse and 100% used debuggers frequently. Also, like many other experimental studies before ours, we asked volunteers to our undergraduate and graduate students at Polytechnique Montréal to participate in our experimental study. After sending a general call for volunteers, 12 students volunteered. They were all experts or advanced developers (70%). They all used IDEs (70%) and debuggers (60%) frequently.

We provided participants with two instructions documents. The first document was an experiment tutorial[11] which explained how to install and configure all tools to perform a warm-up task and the experimental study. We also used the warm-up task to confirm that the participants' environment were correctly configured and that the participants understood the instructions. The warm-up task was described using a video to guide the participants. We make available this video on-line[12].

The second document presented the five issues with a description and some piece of information to reproduce the faults. To reduce the participants' effort to reproduce the faults, we offered videos demonstrating step-by-step how to reproduce the faults. We also provided the participants with an electronic form to report whether they were tired or not at the end of the experiment.

For this experimental study, we used Eclipse Mars 2 and Java 8, the SDI and its Swarm Debug Tracer plug-in, and two Java projects: a small Tetris game for the warm-up task and JabRef v3.2 for the experimental study. All participants received the same workspace, provided by our artifact repository.

After installing the environment (Eclipse and the SDI), each participant executed the warm-up task. These tasks consisted of starting a debugging session, toggling a breakpoint, and debugging a Tetris program to locate a given method. After the warm-up task, each participant executed debugging sessions to find the location of faults described in the five selected issues. We did not set a time constraint but suggested 20 minutes per fault. We asked participants to control their fatigue, asking them to go to the next task if they felt tired while informing us of this situation in their reports. Finally, each participant

[10]https://www.freelancer.com/
[11]<http://swarmdebugging.org/publications/experiment/tutorial.html>
[12]<https://youtu.be/U1sBMpfL2jc>

filled a report to provide their answers and other information, whether they completed the tasks successfully or not.

All services were available on our server[13] during these debugging sessions and the experimental data were collected in the course of 8 days. We also collect the video capture for the participants. The experiment tutorial contained the instruction to install and set the OBS (Open Broadcaster Software), an open source system for live streaming and recording[14]. Participants were asked to provide the video captured during the experiment. We had 28 recorded videos, providing more than **6 hours** of effective developer's activities. We had 38 debugging sessions by 20 developers, 207 collected breakpoints and more than 6000 invocations.

We now report the results of our analyses to answer our research questions.

5.2.2 RQ1: How much time do developers spend to toggle the first breakpoint?

Analysing each screencast recording, we collect the following information:

- Start Time (ST): the effective time when a developer starts a task.
- Time of First Breakpoint (FB): the time when a developer toggles the first breakpoint.
- End time (T): the effective time when a developer finishes a task.
- Elapse End time (ET): $ET = T - ST$
- Elapse Time First Breakpoint (EF): $EF = FB - ST$

In order to compare different task and developer, we normalize for the moment of first breakpoint (MFB):

$$MFB = \frac{EF}{ET} \tag{5.2}$$

MFB shows how much time a developer spend to toggle the first breakpoint comparing to the total elapse task. Calculating MFB to each video, we found that in average, **developers spend 27% of task time to toggle the first breakpoint** (stddev=17%). This result shows that developers spend about 1/4 of the time to locate where to toggle the breakpoint. It means that toggle breakpoint is not an easy task and developers may need tools that could assist them in locating the place to toggle breakpoints. However, we must

[13]<http://server.swarmdebugging.org>
[14]https://obsproject.com

first understand the common locations where developers toggle breakpoints.

5.2.3 RQ2: On what kind of statement do developers toggle their breakpoints?

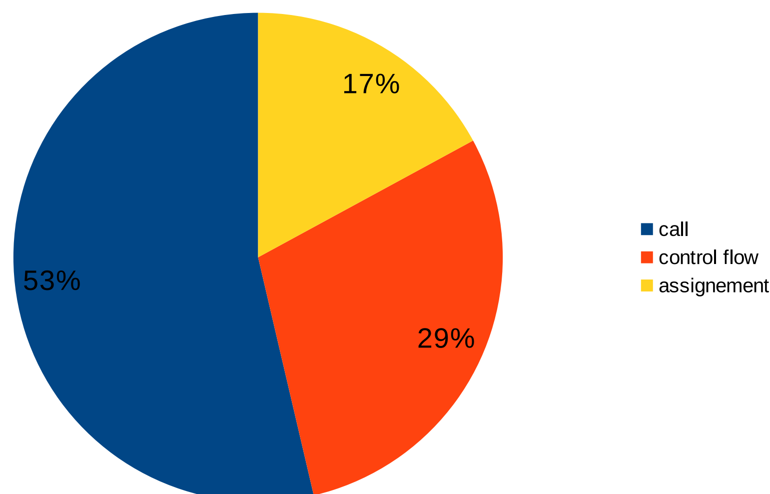
Analysing the code statements for the breakpoints, we can classify what kind of statement the developers toggled their breakpoints. We consider the following kind of statements[15]: *call*, *if-statement*, *assignment*, *return* and *while-loop*. Follow this categories, we analysed each breakpoint and classified it, resulting Table 5.2.

Table 5.2: Breakpoints by kind of statement

Statement	Number of Brekpoints	%
call	111	53
if-statement	39	19
assignment	36	17
return	18	10
while-loop	3	1

Our results show that 53% (111/207) of breakpoints was toggled on call method statements and only 1% (3/207) on while statements. After grouping *if-statement*, *return* and *while-loop* like *control flow* statements, we can observe in Figure 5.8 that 29% of breakpoints are on flow statements. A possible interpretation is that developers prefer *call* statements because they would like to analyze the software state before coming in a method. This result can be useful, for instance, to recommendation systems, increasing a rank to breakpoints on call statements.

Figure 5.8: Breakpoints by kind of statement - call, control flow and assignment



Source: from author

[15]https://en.wikipedia.org/wiki/Statement_(computer_science)

5.2.4 RQ3: Do developers toggle breakpoints in the same place?

We investigated each breakpoint to determinate if there are breakpoints at the same location (type and code line number), analyzing breakpoints in the same tasks and different tasks. Thus, we grouped all breakpoints by **task**, and we counted how many breakpoints was toggled on the same line of code several times for each task, producing Table 5.3. Analysing that table, **we found 39 breakpoints in the exactly same line of code for the same task** toggled by different developers. It is clear that **toggled breakpoints could be useful to other developers who are working in the same task**, as a reopened task or a regression task.

Table 5.3: Breakpoints in the same line of code by task

Task	Type	Line	# same line
0318	AuthorsFormatter	43	5
0318	AuthorsFormatter	131	3
0667	BasePanel	935	2
0667	BasePanel	969	3
0667	JabRefDesktop	430	2
0669	OpenDatabaseAction	268	2
0669	OpenDatabaseAction	433	4
0669	OpenDatabaseAction	451	4
0993	EntryEditor	717	2
0993	EntryEditor	720	2
0993	EntryEditor	723	2
0993	BibDatabase	187	2
0993	BibDatabase	456	2
1026	EntryEditor	1184	2
1026	BibtexParser	160	2

After that, we analyzed if a type had breakpoints for different tasks. Thus, we grouped all breakpoints by **type**, and we counted how many breakpoints was toggled on the type for different tasks, putting "1" if a type had a breakpoint, producing Table 5.4. Furthermore, we counted the number of breakpoints by type and how many developers toggle breakpoints on a type. Analysing that table, **we found ten types that received breakpoints in different tasks by different developers**. For instance, the type **OpenDatabaseAction** had 19 breakpoints in 3 of 5 tasks by 13 different developers. It is clear that **toggled breakpoints could be useful to new maintenance tasks**.

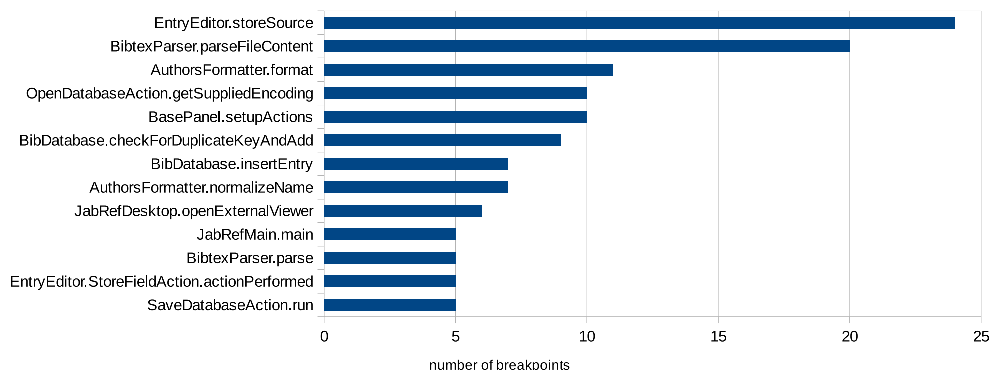
Finally, we count how many breakpoint are in the same method. We found 37 methods that received at least two breakpoints, and 13 methods (Figure 5.9) that received **5 or more breakpoints** during different tasks by different developers. Reviewing that

Table 5.4: Breakpoints by type in different tasks

Type	0318	0667	0669	0993	1026	Breakpoints	Dev Diversity
SaveDatabaseAction	0	0	1	1	1	7	2
BasePanel	1	1	1	0	1	14	7
JabRefDesktop	1	1	0	0	0	9	4
EntryEditor	0	0	1	1	1	36	4
BibtexParser	0	0	1	1	1	44	6
OpenDatabaseAction	0	0	1	1	1	19	13
JabRef	1	1	1	0	0	3	3
JabRefMain	1	1	1	1	0	5	4
URLUtil	1	1	0	0	0	4	2
BibDatabase	0	0	1	1	1	19	4

method aggregation, **we conclude that there are methods that are addressed by many breakpoints in different tasks by several developers**, showing a clear opportunity to use those methods like good candidates for new debugging sessions.

Figure 5.9: Methods with 5 or more breakpoints



Source: from author

5.2.5 RQ4: How effective is co-breakpoint for breakpoint prediction?

The result of **RQ3** shows that breakpoints are sometimes toggled on the same location even of different tasks and–or by different developers. It means that breakpoints previously toggled by a developer could help another developer. To figure out whether this assertion is true, we introduce the concept of co-breakpoint (Section 4.4) and use it to access the effectiveness of using debugging activities to predict breakpoint (sections 4.5 and 5.2.6).

5.2.6 Results and Discussions

We report our results on the use of co-breakpoint for breakpoint prediction by following the process described in Section 4.5. As our experiment involved both students and professional freelancer developers, the way the two kinds of participants debug programs may vary. Professional freelancer developers may tend to carefully and methodically toggle breakpoints while students may not. Thus, we run our prediction approach on the whole dataset, then on only freelancer dataset in one and, and only on students dataset on another hand. This distinction allows us to access whether the debugging activities of one kind of participant could be more "rich" for breakpoint prediction than other. We also predict breakpoint at two level of granularity: class level and method level. The prediction at class level aims to recommend the class that developer need to toggle breakpoint to achieve the task. The prediction at method level recommends the method that developer need to toggle breakpoint to achieve the task.

Table 5.5: Breakpoint Prediction Results

		Class Level	**Method Level**
Freelancer	**Precision**	16.5	15.8
	Recall	62.9	34.7
Students	**Precision**	20	37.5
	Recall	40	50
All	**Precision**	15.5	21.4
	Recall	61.8	35.3

Table 5.5 shows the precision and recall of prediction for freelancer/students and class/method levels. Overall, it indicates that we can use co-breakpoints to recommend the class where a developer could toggle the breakpoint with the precision of 15% and recall of about 62%. We can recommend method with 21% precision and about 35% recall. When looking into the accuracy of prediction using freelancer and students data, we can observe that prediction is better at class level with freelancer data while level method prediction is better with students dataset.

Our results show the potential of debugging activities for breakpoint prediction. However, the low precision of prediction would be due to the small dataset as we have only few breakpoint collected during our experiment. We think that more dataset would improve the accuracy of the prediction.

This promising results of breakpoint prediction show the usefulness of collecting and sharing debugging activities to assist developers during maintenance activities.

5.2.7 Threats to Validity

This section discusses the threats to the validity of our results.

Construct Validity threats is related to the metrics used to answer our research questions. We mainly used breakpoint counts as a precise measure. However, we considered the breakpoints collected by our swarm debugging infrastructure (SDI). Any issue regarding the collection of breakpoints with SDI would affect our results. To mitigate these threats, we collected both SDI data and video captures of the screen of participants. We compared information extracted from the videos with the data collected by SDI and found that the breakpoints collected by SDI are exactly those toggled by developers.

Conclusion Validity threats concerns the violations of the assumptions of the statistical tests, and how diverse is the collected data. We reported results in terms of percentages of breakpoints toggled for different kinds of statements, and the common breakpoints toggled on class/method for the same and different tasks. We did not perform any statistical analysis to answer our research questions. Thus, our results do not suffer from any statistical assumptions. We do not claim any causation relationship between the number/percentage of breakpoints, the kind of statements, and the tasks or developers.

Internal Validity threats are related to the tools used to collect the data and the subject systems. We use SDI and any issue with SDI would affect our results. However, as we validated the collection of breakpoints using the videos, the threat related to SDI is mitigated. We also used videos to identify when developers start and finish the tasks. Thus, the threat regarding the SDI is addressed by using the video capture of the screen. We use only one system in our study (*i.e.,* JabRef). We performed our study on a single system because we needed to have enough data points from a single system to assess the effectiveness of breakpoint prediction. We should collect more data on other systems and check whether the system used can affect our results. Each developer (*e.g.,* freelancer) performed more than one task on the same system. It is possible that a participant may have become familiar with the system after performing earlier tasks and would be knowledgeable enough to toggle breakpoints when performing later tasks. However, we didn't observe any significant difference in performance when comparing the performance of same developers for the first and last task.

External validity threats concern the possibility to generalize our results. We share our data and scripts at http://swarmdebugging.org/publications/icsme2016. Further studies with different sets of tasks and different participants are required to verify our

results and make our findings more general.

5.3 Experiment 3 - supporting maintenance tasks using shared debugging visualisations

In this section, to assess the usefulness of GV, we conducted a qualitative study with 22 professional developers and a controlled experiment with 14 professional developers. We report qualitative and quantitative studies showing the benefits of sharing debugging data.

5.3.1 Experiment design

This section presents the design of our experiment, which aims to address the following tworesearch questions:

RQ1: Is Global View useful to support software maintenance tasks?

RQ2: Does sharing and visualizing debug data support software maintenance tasks?

5.3.2 Experiment setup

To answer the research questions, we proceeded as follows[16].

We had to choose debugging tasks to trigger participants' debugging activities. We chose to ask participants to find the locations of true faults in an independent, open-source program. We selected JabRef[17] as target program, which is an open-source bibliography reference manager developed in Java. We chose JabRef because it has faults publicly reported in its issue tracker, its domain was easy to understand by the participants, and it has modules with some relative independent regions. We picked five faults reported against JabRef v3.2 in its issue tracker (*i.e.,* issues 993, 1026, 1173, 1235 and 1251) and asked participants to find the locations of the faults.

We performed two different studies: 1) a qualitative evaluation using GV on a browser; and 2) a controlled experiment on bug location tasks, using GV integrated into Eclipse.

The qualitative evaluation consisted of a set of questions about JabRef issues, using only GV on a regular web browser (without JabRef's source code). The participants

[16] Artifacts available at <http://swarmdebugging.org/publications/vissoft2016>.

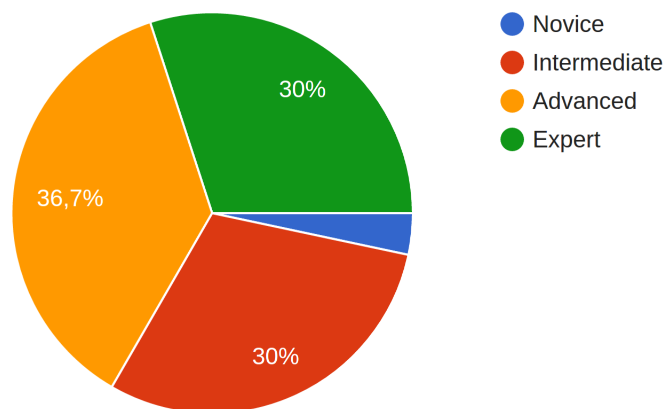
[17] <http://www.jabref.org/>

were asked to identify the “type" in which the issue was located for Issue 318, Issue 667, and Issue 669, using only GV. An explanation was required for each answer. In addition to providing information about the usefulness of GV for tasks' comprehension, this phase helped to familiarize the participants with GV.

The controlled experiment was a true bug location task, in which some developers were asked to find the location of a fault using the GV integrated into their Eclipse IDE. We divided the participants into two groups: control group (7 participants) and experimental group (6 participants). Participants from the control group performed tasks 993 and 1026 **without using GV**, while those from the experimental group performed the tasks using GV.

To reproduce a realistic industry scenario, we recruited 30 professional freelancer developers[18]. Among them, they have on average six years of experience in software development (stdev=4); on average 4.8 years of Java experience (stdev = 3.3), and 67% are advanced or experts on Java (figure 5.10); 77% male and 23% female; 97% used Eclipse. Among of this professionals, 23 participated in the qualitative evaluation, and six participated in the bug location using GV in Eclipse.

Figure 5.10: Java expertise



Source: from author

After the profile survey, we provided artifacts to support the two phases. For the phase one, we provided an electronic form with instructions to follow and questions to answer. The GV was available at <http://server.swarmdebugging.org/>. For the phase two, we provided participants with two instruction documents. The first document was an experiment tutorial[19] that explained how to install and configure all tools to perform a warm-up task and the experimental study. We also used the warm-up task to confirm

[18]https://www.freelancer.com/
[19]<http://swarmdebugging.org/publications/experiment/tutorial.html>

that the participants' environment were correctly configured and that the participants understood the instructions. The warm-up task was described using a video to guide the participants. We make available this video on-line[20]. The second document was an electronic form to collect the results and other evaluations made using the integrated GV.

For this experimental study, we used Eclipse Mars 2 and Java 8, the SDI with GV and its Swarm Debug Tracer plug-in, and two Java projects: a small Tetris game for the warm-up task and JabRef v3.2 for the experimental study. All participants received the same workspace, provided by our artifact repository.

In the qualitative evaluation, the participants answered the questions directly in the electronic form. They used the GV available online (<http://server.swarmdebugging.org/>) with collected data for JabRef issues' 318, 667, 669.

In the controlled experiment, after installing the environment (Eclipse and the SDI), each participant executed the warm-up task. This task consisted of starting a debugging session, toggling a breakpoint, and debugging a Tetris program to locate a given method. After the warm-up task, each participant executed debugging sessions to find the location of faults described in the five selected issues. We set a time constraint of one hour. We asked participants to control their fatigue, asking them to go to the next task if they felt tired while informing us of this situation in their reports. Finally, each participant filled a report to provide their answers and other information, whether they completed the tasks successfully or not, commenting on the usefulness of GV during each task.

All services were available on our server[21] during these debugging sessions and the experimental data were collected in the course of 3 days. We also collected the video capture from the participants. The experiment tutorial contained the instruction to install and set the OBS (Open Broadcaster Software), an open source system for live streaming and recording[22]. Participants were asked to provide the video captured during the experiment. We obtained ten recorded videos, providing more than **3 hours** of effective developer's activities.

Next, we now report the results of our analyses to answer our research questions.

[20]<https://youtu.be/U1sBMpfL2jc>
[21]<http://server.swarmdebugging.org>
[22]https://obsproject.com

5.3.3 Is Global View useful to support software maintenance tasks?

During the qualitative evaluation, we asked the participants to analyze the graph generated by GV to identify the type of the location of each bug, **without reading the task description or looking at the code**. The GV generated graph had invocations collected from previous debugging sessions. These invocations were generated during "good" sessions (the bug was found) and "bad" sessions (the bug was not found). We analyzed results obtained for task 318, 667 and 699, comparing the number of participants that were able to propose a solution and the correctness of the solutions.

For the task 318 (Figure 5.11), 95% of participants (22/23) were able to suggest a "candidate" type for the location of the fault, just by using the GV view. Among these participants, **52% (12/23) suggested correctly *AuthorsFormatter* as the problematic type**.

Figure 5.11: GV by Task 0318

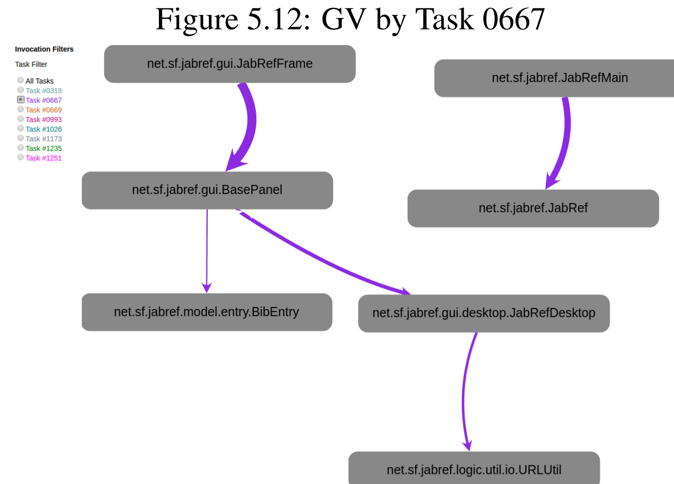
Source: from author

For the task 667 (Figure 5.12), 95% of participants (22/23) were also able to suggest a "candidate" type for the problematic code, just by using the GV view. Among these participants, **31% (7/23) suggested correctly that *URLUtil* was the problematic type**.

Finally, for the task 669 (Figure 5.13), again 95% of participants (22/23) were able to suggest a "candidate" for the type of the problematic code, just by looking at the GV view. However, none of them (*i.e.,* **0% (0/23)**) provided the correct answer, that *OpenDatabaseAction* was the problematic type.

RQ1 asked if GV is useful to support software maintenance tasks. We answered that **87% of participants agreed that GV is useful or very useful (100% at least some usefulness)** through our qualitative study (Figure 5.16), and **75% of participants claimed that GV is useful or very useful (100% at least some usefulness)** on the task result survey after true bug location tasks (Figure 5.16). Furthermore, several participants' feedbacks support our answers.

Figure 5.12: GV by Task 0667



Source: Source: from author

Analysing those data, our results suggest that 1) GV is useful to support software maintenance tasks, and 3) there is a direct relation between the quality of collected session data and participants' performance to create hypotheses about the location of an issue.

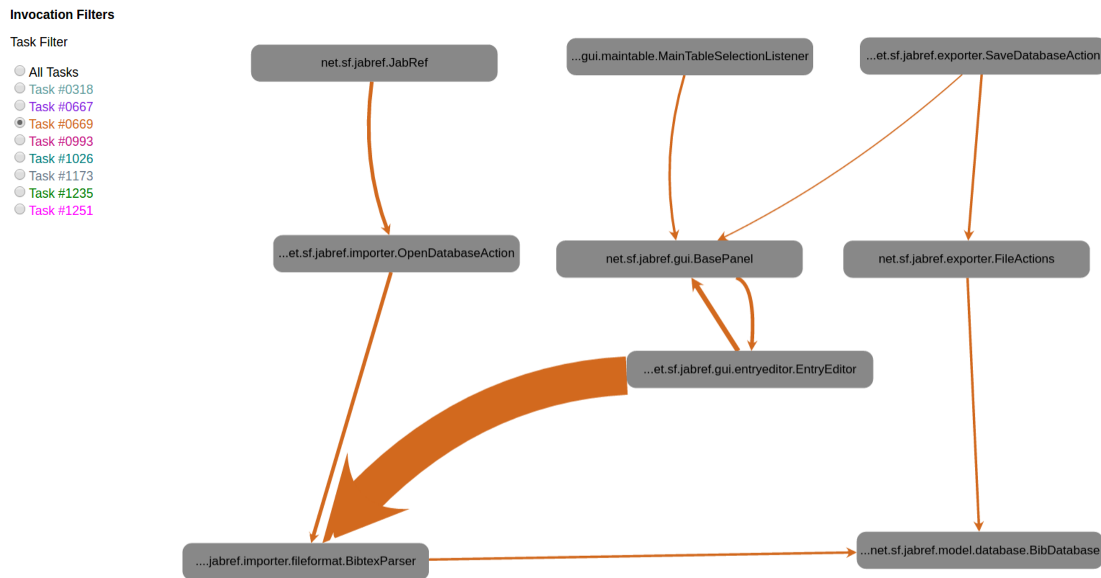
5.3.4 Does sharing and visualizing debug data support software maintenance tasks?

RQ2 was about sharing and visualizing debugging data to support software maintenance and evolution tasks. We analyzed each video recording and searched for evidence of GV utilization during genuine bug locations tasks. Our controlled experiment showed that **100% of participants** used GV to support their tasks (video recording analysis), navigating, reorganizing and especially diving into the type double-clicking on a selected type, as we can observe as an example in Figure 5.18.

5.3.5 Participants' Feedback

As most visualization techniques proposed in the literature, ours is a prototype used as a proof of concept but with both intrinsic and accidental advantages and limitations. While intrinsic advantages/limitations pertain to the visualization itself and our design choices, accidental advantages/limitations concern our implementation. During our experiment, we collected the participants' feedback on our visualization and discussed both intrinsic and accidental advantages/limitations as reported by the participants. We will return on some of the limitations in the next section that describes threats to the validity of our experiment. Finally, we report the general feedback from one of the participant.

Figure 5.13: GV by Task 0669



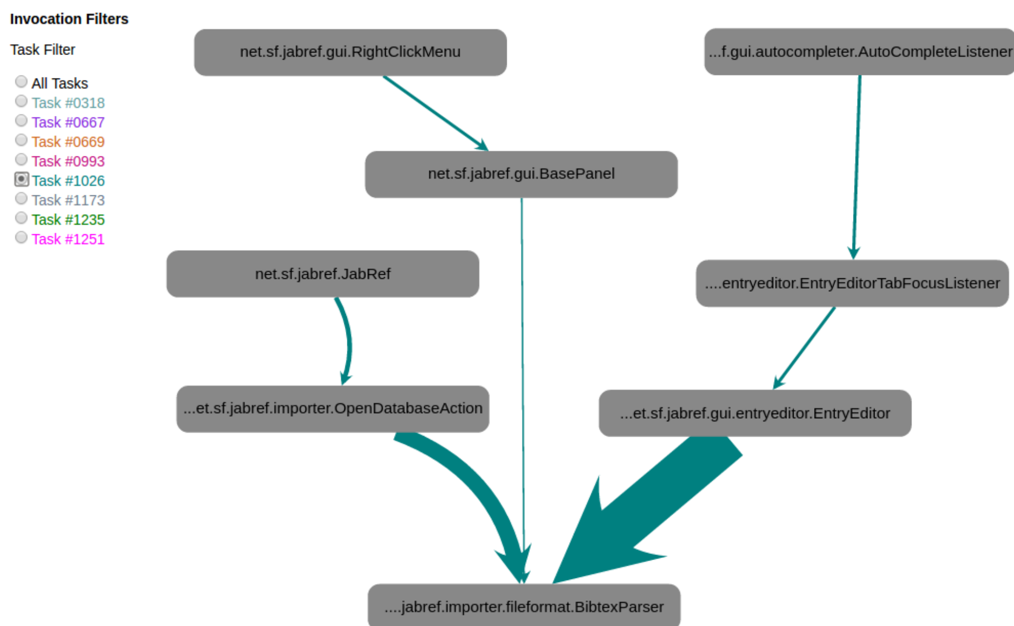
Source: from author

Visualisation of Debugging Paths Several participants commented our visualization for presenting useful information related to the classes and methods followed by other developers during debugging for some task. In particular, one participant reported that "[i]t seems a fairly simple way to visualize classes and to demonstrate how they interact.".

Effort in Debugging Three participants additionally mentioned that our visualization shows where developers spent their debugging effort and where understanding "bottle-necks" are. In particular, one participant wrote that our visualization "allows the developer to skip several steps in debugging, knowing from the graph where the problem probably comes from."

Location One participant commented that "the location where [an] issue occurs is not the same as the one that is responsible for the issue." We are well aware of this difference between the location where a bug occurs, for example, a null-pointer exception, and the location of the source of the bug, for example, a constructor where the field is not initialized. However, we build our visualization on the premise that developers can share their debugging activities for that very specific reason: by sharing, they could readily identify the source of a bug rather than only the location where it occurs. We plan to perform further studies to assess the usefulness of our visualization to achieve or not our premise.

Figure 5.14: GV by Task 1026



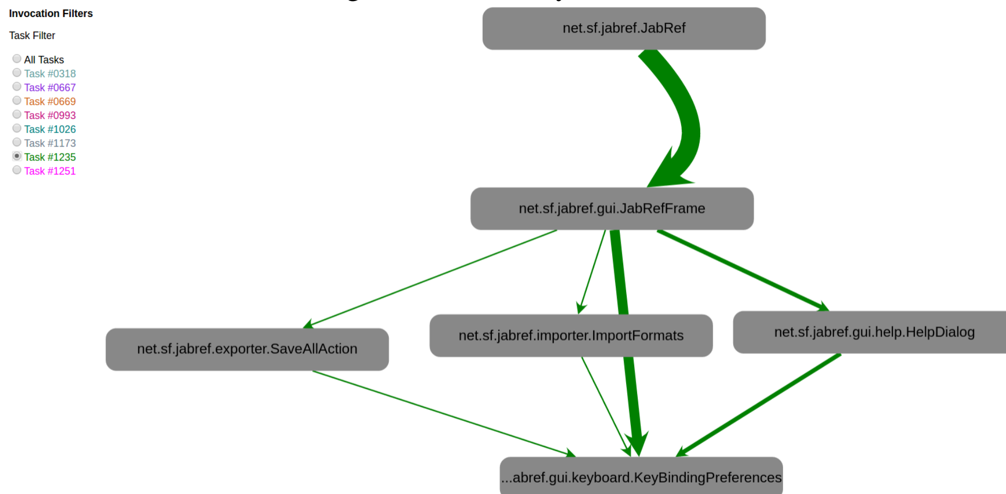
Source: from author

Scalability Several participants commented on the possible lack of scalability of our visualization. Graph are well known to be not scalable, so we are expecting issues with larger graphs (INTERNATIONAL CONFERENCE ON BIG DATA AND SMART COMPUTING, 2015, 2015). Strategies to mitigate these issues include graph sampling but also clustering.

Presentation Several participants also commented on the (relative) lack of information brought by the visualization, which complementary to the issue of scalability. One participant commented on the difference between the graph showing the developers' paths and the relative importance of classes during execution. Future work should seek to combine both information on the same graph, possibly by combining size and colors: size could relate to the developers' paths while colors could indicate the "importance" of a class during execution.

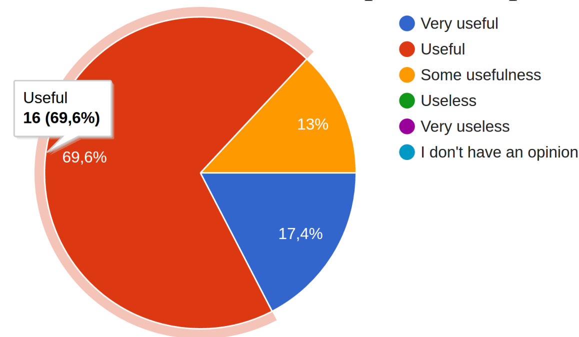
Evolution One participant commented that the graph is relevant for one version of the system at hand but that, as soon as some changes are performed by a developer, the shown paths may become irrelevant. We agree with the participant and accept this criticism because our visualization was thought for one version. We will explore in future work how to handle evolution and perform the minimal changes to the graph.

Figure 5.15: GV by Task 1235



Source: from author

Figure 5.16: GV usefulness - experimental phase one



Source: from author

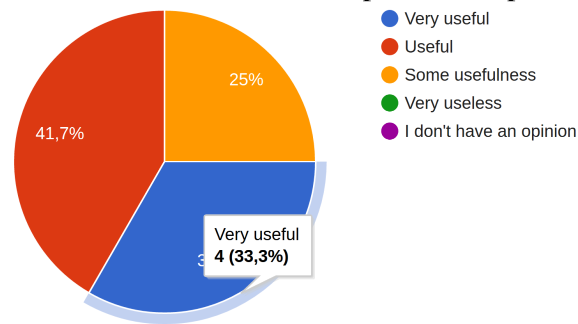
Trap One participant warned that our visualization could lead developers into a "trap" if, for some reasons, all developers whose paths are being displayed followed the "wrong" paths. We agree with the participant and accept this risk because we assume that developers would not, en masse, choose the wrong paths.

Understanding One participant reported that the visualization alone does not bring enough information to understand fully the task at hand. We accept this limitation because our visualization is built to be complementary to other views available in the IDE.

Reducing Code Complexity One participant discussed the use of our visualization to reduce code complexity for the developers by highlighting its main functionalities.

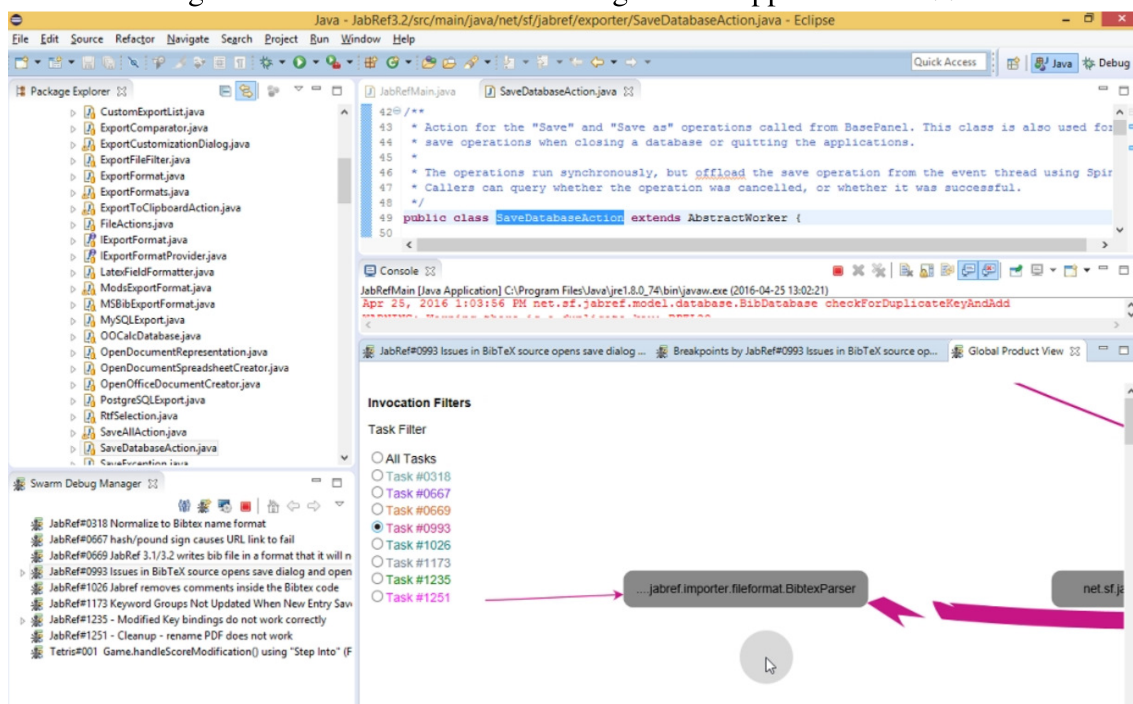
Complementing Differential Views Another participant contrasted our visualization with Git Diff and mentioned that they complement each other well because of our visual-

Figure 5.17: GV usefulness - experimental phase two



Source: from author

Figure 5.18: Video on Dev520 using GV to support the task 993.



Source: from author

ization “[a]llows to quickly see where the problem probably has been before it got fixed.” while Git Diff allows seeing where the problem was fixed.

Highlighting Refactoring Opportunities Finally, a third participant suggested that the larger node could represent classes that could be refactored if they also have many bugs to simplify future debugging sessions for developers.

Presentation Several participants commented on the presentation of the information by our visualization. Most importantly, they remarked that identifying the location of the bug was difficult because there was no distinction between faulty and non-faulty classes. In the future, we will assess the use of icons and–or colors to identify faulty classes/methods.

Others commented on the lack of caption describing the various visual elements. Although this information was present in the tutorial and questionnaires, we will add it also into the visualization, possibly using tool tips. One participant added that more information, such as “execution time metrics [by] invocations” and “failure/success rate [by] invocations” could be valuable information. We plan to perform other controlled experiments with such additional information to asses its impact on developers’ performance. Finally, one participant mentioned that arrows would sometimes overlap, which points to the need for a better placement algorithm for the graph nodes in our visualization.

Navigation One participant commented that the visualization does not help developers navigating between classes whose methods have a low cohesion. It should be possible to show in different parts of the graph the methods and their classes if these methods have low cohesion so as to avoid large nods in the graph that are accidentally large (because of the sheer numbers of their methods). We plan to modify the graph visualization to have a “method-level” view whose node could be methods and–or clusters of methods (unrelated to their classes).

5.3.6 General Feedback

Three participants left general feedback regarding their experience with our visualization in the “Describe your debugging experience” free text part of the questionnaire. All three participants provided positive comments. We report here one of the three comments:

> *It went pretty well. In the beginning, I was at a loss, so just was looking around for some time. Then I opened the breakpoints view for another task that was related to file parsing in the hope to find some hints. And indeed I’ve found the BibtexParser class where the method with the most number of breakpoints was the one where I later found the bug. However, only this knowledge was not enough, so I had to study the code a bit. Luckily, it didn’t require too much effort to spot the problem because all the related code was concentrated inside the parser class. Luckily I had a BibTeX database at hand to use it for debugging. It was excellent.*

This comment highlights the benefits of our visualization and confirm that our premise may be correct and that developers may benefit from others’ debugging sessions. It encourages us to pursue our research work in this direction and perform more experiments and improve our visualization.

5.3.7 Threats to Validity

Despite its promising results, there exist threats to the validity of our experiment that we discuss in the following.

Construct Validity Threats are related to the metrics used to answer our research questions. We mainly used breakpoint locations, which is a precise measure. Yet, as we located breakpoints using our Swarm Debug Infrastructure (SDI) and visualization, any issue with it would affect our results. To mitigate these threats, we collected both SDI data and video captures of the participants' screens and compared the information extracted from the videos with the data collected by the SDI. We observed that the breakpoints collected by the SDI are exactly those toggled by the participants.

We ask participants to self-report on their efforts during the tasks, levels of experience, etc. through the questionnaires. Consequently, it is possible that the answer to represent their *real* efforts, levels, etc. We accept this threat because questionnaires are the best means to collect data about participants without incurring a high cost. Construct validity could be improved in future work by using instruments to measure effort independently, for example, but such instruments would lead to more time- and effort-consuming experiments.

Conclusion Validity Threats concern the relations found between independent and dependent variables. In particular, they concern the assumptions of the statistical tests performed on the data and how diverse is the data. We did not perform any statistical analysis to answer our research questions, so our results do not depend on any statistical assumption.

Internal Validity Threats pertain the tools used to collect the data and the subject systems and if the collected data is sufficient to answer the research questions. We collected data using our visualization. We are well aware that our visualization does not scale for large systems but, for JabRef, it allows participants to share paths during debugging and researchers to collect relevant data, including shared paths. We plan to further study our visualization to identify possibilities to improve it so that it scales up to large systems.

Each participant performed more than one task on the same system. It is possible that a participant may have become familiar with the system after performing an earlier task and would be knowledgeable enough to toggle breakpoints when performing later tasks. However, we did not observe any significant difference in performance when comparing the performance of a same participant for the first and last task. Therefore, we

accept this threat but still plan for future studies with more tasks on more systems.

External Validity Threats are about the possibility to generalize our results. We use only one system in our study (JabRef) because we needed to have enough data points from a single system to assess the effectiveness of breakpoint prediction. We share our data and code on-line[23]. We should collect more data on other systems and check whether the system used can affect our results.

Moreover, we should collect more data on different *versions* of some systems, to assess the benefit of our visualization in time during software evolution, which is the most important of all developers' tasks. Future studies with different sets of tasks and different participants are also necessary to verify our results and make our findings more general.

The participant to our controlled experiment were free-lancers that were paid to perform the tasks, albeit small amounts of money **CAD$five by the phase one and CAD$10 per task on the phase two**. Paying participants is a means to thank them for their time and effort but also introduce the threat that the participants try to please the researchers to obtain their payments, through the Pygmalion/Rosenthal effect. We are planning future studies with volunteer students and developers to assess the impact of this threat on our results. For this experiment, we accept the threat because the amounts of money are small enough not to entice unduly participants.

5.4 Final remarks

This chapter presented three studies in other to evaluate the effectiveness of the Swarm Debugging Infrastructure, showing the data collected by SDI could be use to understand interactive debugging. First, we conducted an experiment with three real maintenance task performed by seven developers on JabRef system. The main findings are: (1) there is not a correlation between the numbers of invocations and elapse task time ($\rho = -0.039$); (2) there is no correlation between numbers of breakpoints and elapse task time ($\rho = 0.093$); (3) developers follow different debugging patterns (4) there is no relation between numbers of breakpoints and expertise; (5) whether developers toggle breakpoints carefully, they complete tasks faster than developers who toggle breakpoints quickly.

Next, we conducted a study and collected debugging activities of 20 developers when performing realistic maintenance tasks on a Java open-source system. We study how

[23] http://swarmdebugging.org/publications/vissoft2016

developers toggled breakpoints and observed that developers spent among of time (27%) to toggle the first breakpoint. Moreover, developers who carefully toggled breakpoints are more efficient (in term time spent to resolve the task) than those who toggle breakpoint early. By analyzing the target statements of breakpoints, we found that breakpoints are usually toggled on call statements (53% of breakpoints) and that breakpoints are sometimes toggled on the same location even for different tasks and–or by different developers. This observation calls for the investigation whether breakpoints previously toggled by a developer could help another developer. Thus, we tested the concept of co-breakpoint and used it for breakpoint prediction. We predicted breakpoint at class and method level and achieved the precision and recall of 15% and 62%, and 21% and 35%, respectively for class and method level. Our results show that collecting and sharing debugging activities could help to study and improve software maintenance and evolution.

Finally, we assessed the usefulness of GV for conducting a qualitative study with 23 professional developers and a controlled experiment with 13 professional developers. We answered the following research questions. RQ1 asked if GV is useful to support software maintenance tasks. We answered that **87% of participants agreed that GV is useful or very useful (100% at least some usefulness)** through our qualitative study. RQ2 was about sharing and visualizing debug data support software maintenance and evolution tasks. Our controlled experiment showed that **100% of participants** used GV to support their tasks (video recording analysis), and **75% of participants claimed that GV is useful or very useful (100% at least some usefulness)** in the survey. Furthermore, several participants' feedbacks support our answers. Thus, we proposed a visualization that development teams can use to share debugging data and reported qualitative and quantitative data showing the benefits of sharing debugging data. Besides, we also reported the feedback of the participants who took part of our controlled experiments and showed that they identified both essential and accidental advantages and limitations to our visualization.

In conclusion, these three studies presented three examples how SD and SDI could be used to investigate debugging phenomena and create a recommendation system and a visualization. The results showed that SD is an approach that ables to support and improve debugging activities and researches on interactive debugging.

6 CONCLUSION

Understanding software and fixing software defects are complex, tedious and time consuming activities. To address these laborious tasks, developers often exploit applications through debugging. As a result, this process produces a lot of information about the system context. This information is, however, usually lost after the end of the debugging sessions. In this thesis, we presented a new approach named Swarm Debugging (SD) aiming to collect, share and retrieve information from debugging sessions. SD uses developers' cooperative effort (FUGGETTA, 2000; STOREY et al., 2014) to capture and share knowledge, collecting iterations that are usually discarded in traditional debugging tools. It allows developers to find breakpoints and starting points and share their experiences on software projects transparently. Focusing on sensitive session context, each SD session captures only the paths covered intentionally, driven by real developers' issues.

In order to provide support for our approach, we developed the Swarm Debug Infrastructure (SDI), an open-source infrastructure integrated into Eclipse, to collect and share fine-grained data about developers' interactive debugging activities. The SDI collects data in the background during debugging activities without affecting the performance of the IDE. SDI stores data on a remote server using an asynchronous execution and, thus, does not suffer from performance or memory issues as it occurs with omniscient debuggers (POTHIER; TANTER, 2009) or tracing-based approaches (OHMANN; LIBLIT, 2013). Moreover, our approach is not limited to a single language or architecture, requiring only that a debugging tool performs simple HTTP requests through a RESTful API, allowing that new clients were implemented easy.

SD shows as a significant feature the division of the software complexity problem. In traditional approaches, as the JIVE approach, all data are used to construct visualizations and search. Moreover, when the session finishes all data are lost. On the other hand, Swarm Debugging saves session information, and developers can divide massive projects on multiple diagrams, having a fine control over exploring relevant, important software areas. This feature improves software comprehension. SD is a complementary approach for JIVE (GESTWICKI; JAYARAMAN, 2005), addressing the sharing of information and visual scalability issues.

We conducted experimental studies to assess the effectiveness of the SD in collecting data relevant for comprehension studies and debugging. Using data provided by SDI we performed empirical studies with professional (and freelancer) developers, and we

could answer several research questions on which we observe that (1) developers spent time (about 27%) to toggle the first breakpoint; (2) developers that carefully toggled breakpoints are more efficient (in terms of time spent to resolve the task) than those who toggled breakpoints earlier; (3) breakpoints are usually toggled on call statements (53% of breakpoints) and breakpoint are sometimes toggled on the same location even for different tasks and–or by different developers. These observations lead to the investigation of whether breakpoints previously toggled by a developer could help another developer, resulting in the concept of co-breakpoint. In conclusion, our results show that Swarm Debugging could help and improve software maintenance and evolution.

6.1 Summary of contributions

This research has two main contributions. First, a new approach to collect, share and retrieve information from debugging sessions named Swarm Debugging. Second, an infrastructure to support our approach (Swarm Debug Infrastructure), providing several visualisations and searching tools.

SDI is an open and freely available infrastructure that SE researchers can use to perform new empirical studies about debugging and–or software static and dynamic analysis.

Developers can use SDI to record their debugging patterns to identify debugging strategies that are more efficient in the context of their project, allowing them to improve their debugging skills.

Moreover, they can also use SDI to store and share project information, such as breakpoints and–or invocations. Developers can share their debugging activities, such as breakpoints and–or invocations to improve collaborative work and ease software maintenance. While developers usually work on specific tasks, there are sometimes re-open issues and–or similar tasks that need to understand or toggle breakpoints on the same entity. Thus, using breakpoints previously toggled by a developer could help to assist another developer working on similar tasks. For instance, the breakpoint search tools can be used to retrieve breakpoints from previous debug sessions, which could help speed up a new debugging session, providing developers with valid starting points. Therefore, the breakpoint search tool can decrease the time spent to toggle a new breakpoint.

Debugger's developers can use SDI to understand IDE users' behaviors and requirements. This knowledge base is important to create new tools, using novel data-

mining techniques, to integrate different data sources. SDI provides a transparent framework for developers to share debugging information, creating a collective intelligence about their software projects.

The understanding of how developers perform debugging activities and–or debugging patterns is an example of debugging activities collected by SDI. In addition to this perspective, the data collected by SDI can be useful to assess the developers' interest and knowledge of the code: when developers toggle a breakpoint in a program element, the element seems relevant to the task at hand. By collecting debugging activities, we can know the program elements which raised interest in a developer for solving a task. Thus, these debugging activities could help to assess the developers' knowledge of the code.

Last but not least, educators could leverage SDI tools to teach interactive debugging techniques, tracing their students' debugging sessions, and evaluating their performance. Data collected by SDI about debugging sessions of professional development could also be used to educate students, *e.g.,* by showing them examples of good and bad debugging patterns.

6.2 Limitations

Our work has the following limitations:

Versioning Our approach currently support only a single software version.

Platform Despite our model is generic, our approach currently has an implementation that uses Eclipse for Java projects.

Collecting As any knowledge-oriented approach, SD works only if developers collect data about a software project using SDI, improving its coverage as the amount of tasks increase in time. Furthermore, some project's regions could not be debugged, and consequently, they are not collected by SDI. However, this limitation opens opportunities for exploring why some areas are never explored.

Visualisation scaling Despite task filtering and collecting only real paths, our visualizations are not prepared for large-scale systems, and this limitation has to be addressed in future work.

In fact, such limitations are indeed challenges for future work.

6.3 Future work

In future work, we plan to extend the SDI into a full context-aware debugger, adding new ways to share and visualise debugging sessions, implementing SDI Tracer for several languages on Eclipse (as C/C++, PHP, Python, and Ruby), and tracers to Netbeans, IntelliJ IDEA, Visual Studio, Pharo and GDB. We are working to create a tracer for JavaScript, using Firebug and Google Chrome Developer Tools. We also plan to integrate the SDI with a bug tracking system, improve the breakpoint search, associate issues tracking information with breakpoints. Finally, we will perform new empirical studies on developers' debugging activities using the SDI, exploring multi-language behavior about debugging, towards a deep study of debugging activities.

We also intend to implement the recommendation of breakpoints, using the co-breakpoint concept and similarity of tasks. They can be integrated with the graphical UI that can point the developer to the location where to toggle a breakpoint. We will combine debugging data with other "usual" call-graph as well as icons to identify the classes with faults. We will also devise algorithms to change as little as possible the visualization when changes are performed in the code. Finally, we will perform further controlled experiments with more tasks and more systems and different measures. We will also explore other visualization techniques than graphs.

Furthermore, we plan to use SDI in an authentic production environment and survey developers about the usefulness of SDI. We would also ask the opinion of other developers of debugging tools to figure out whether SDI could be beneficial to the community of debugging tools and–or integrated with existing debugging tools and approaches, as Hipikat (CUBRANIC et al., 2005), for example.

Last but not least, the research community can leverage the SDI to conduct more studies to improve our understanding of developers' debugging activities, which could ultimately result in the development of whole new families of debugging tools that are more efficient and–or more adapted to the particularity of each debugging activity. Many open questions are remaining without answers yet, and this thesis is just a first step towards fully understanding and improvements on debugging activities.

REFERENCES

OHMANN, Peter and Liblit, Ben.

AN, L.; KHOMH, F.; ADAMS, B. Supplementary Bug Fixes vs. Re-opened Bugs. In: IEEE INTERNATIONAL WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION, 14TH, 2014. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 2014. p. 205–214.

ARAKI, K.; FURUKAWA, Z.; CHENG, J. A general framework for debugging. **IEEE Software**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 8, n. 3, p. 14–20, May 1991.

ASHOK, B. et al. Debugadvisor: A recommender system for debugging. In: ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, 7TH, 2009. New York, NY, USA: ACM, 2009. p. 373–382.

AYEWAH, N. et al. Using Static Analysis to Find Bugs. **IEEE Software**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 25, n. 5, p. 22–29, sep 2008.

BALL, T.; EICK, S. G. Software visualization in the large. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 29, n. 4, p. 33–43, abr. 1996.

BALL, T.; RAJAMANI, S. K. The slam project: Debugging system software via static analysis. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 37, n. 1, p. 1–3, jan. 2002. ISSN 0362-1340.

BALZER, M.; DEUSSEN, O. Level-of-detail visualization of clustered graph layouts. In: INTERNATIONAL ASIA-PACIFIC SYMPOSIUM ON VISUALIZATION, 6TH, 2007. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 2007. p. 133–140.

BANTELAY, F.; ZANJANI, M. B.; KAGDI, H. Comparing and combining evolutionary couplings from interactions and commits. **Working Conference on Reverse Engineering, 20th, 2013**, IEEE Computer Society, Los Alamitos, CA, USA, v. 00, p. 311–320, 2013.

BARR, E. T.; MARRON, M. Tardis: Affordable time-travel debugging in managed runtimes. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 49, n. 10, p. 67–82, oct. 2014. ISSN 0362-1340.

BLASCIAK, A.; PARETS, G. **System of debugging software through use of code markers inserted into spaces in the source code during and after compilation**.

BOOCH, G. Then a Miracle Occurs. **IEEE Software**, IEEE Computer Society, Washington, DC, USA, v. 32, n. 4, p. 12–14, jul. 2015.

BROOKS, F. P. No silver bullet: Essence and accidents of software engeneering. **IEEE Computer**, IEEE Computer Society, Washington, DC, USA, v. 20, n. 4, p. 10–19, April 1987.

BRUCH, M. et al. Ide 2.0: Collective intelligence in software development. In: WORKSHOP ON FUTURE OF SOFTWARE ENGINEERING RESEARCH, 2010. New York, NY, USA: ACM, 2010. p. 53–58.

BUGDE, S. et al. Global Software Servicing: Observational Experiences at Microsoft. In: IEEE INTERNATIONAL CONFERENCE ON GLOBAL SOFTWARE ENGINEERING, 2008. **Proceedings...** [S.l.]: IEEE, 2008. p. 182–191.

CASERTA, P.; ZENDRA, O.; BODéNèS, D. 3d hierarchical edge bundles to visualize relations in a software city metaphor. In: IEEE INTERNATIONAL WORKSHOP ON VISUALIZING SOFTWARE FOR UNDERSTANDING AND ANALYSIS, 6TH, 2011. Washington, DC, USA: IEEE Computer Society, 2011. p. 1–8.

CECCATO, M. et al. Do Automatically Generated Test Cases Make Debugging Easier? An Experimental Assessment of Debugging Effectiveness and Efficiency. **ACM Transactions on Software Engineering and Methodology**, v. 25, n. 1, p. 1–38, Dec 2015.

CHEN, F.; KIM, S. Crowd debugging. In: FOUNDATIONS OF SOFTWARE ENGINEERING, 10TH, 2015. New York, NY, USA: ACM, 2015. p. 320–332.

CHIC, A.; NIERSTRASZ, O.; GIRBA, T. Towards a moldable debugger. In: WORKSHOP ON DYNAMIC LANGUAGES AND APPLICATIONS, 7TH, 2013. **Proceedings...** New York, NY, USA: ACM, 2013. p. 2:1–2:6.

CHOW, T.; CAO, D.-B. A survey study of critical success factors in agile software projects. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 81, n. 6, p. 961–971, jun. 2008.

COCKBURN, A. **Agile Software Development: The Cooperative Game, Second Edition**. [S.l.]: Addison-Wesley Professional, 2006. 504 p.

CORNELISSEN, B.; ZAIDMAN, A.; DEURSEN, A. V. A controlled experiment for program comprehension through trace visualization. **IEEE Transactions on Software Engineering**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 37, n. 3, p. 341–355, 2011.

CORNELISSEN, B. et al. A Systematic Survey of Program Comprehension through Dynamic Analysis. **IEEE Transactions on Software Engineering**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 35, n. 5, p. 684–702, sep. 2009.

CUBRANIC, D.; MURPHY, G. C. Hipikat: Recommending pertinent software development artifacts. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 25TH, 2003. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2003. p. 408–418.

CUBRANIC, D. et al. Hipikat: A project memory for software development. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 31, n. 6, p. 446–465, jun. 2005. ISSN 0098-5589.

CZYZ, J. K.; JAYARAMAN, B. Declarative and visual debugging in eclipse. In: OOPSLA WORKSHOP ON ECLIPSE TECHNOLOGY EXCHANGE, 2007. **Proceedings...** New York, NY, USA: ACM, 2007. p. 31–35.

DELINE, R. et al. Debugger canvas: Industrial experience with the code bubbles paradigm. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 34TH, 2012. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. p. 1064–1073.

DEMEYER, S.; DUCASSE, S.; LANZA, M. A hybrid reverse engineering approach combining metrics and program visualisation. In: WORKING CONFERENCE ON REVERSE ENGINEERING, SIXTH, 1999. **Procedings...** Washington, DC, USA: IEEE Computer Society, 1999. p. 175–186.

DENEUBOURG, J. L. et al. The self-organizing exploratory pattern of the argentine ant. **Journal of Insect Behavior**, [S.l. : S.n.], v. 3, n. 2, p. 159–168, mar 1990.

ESTLER, H. C. et al. Collaborative debugging. In: IEEE INTERNATIONAL CONFERENCE ON GLOBAL SOFTWARE ENGINEERING, 8TH, 2013. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2013. p. 110–119.

FLEMING, S. D. et al. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 22, n. 2, p. 14:1–14:41, mar. 2013.

FOWLER, M. **Datensparsamkeit**. 2016. Available from Internet: <http://martinfowler.com/bliki/Datensparsamkeit.html>.

FUGGETTA, A. Software process: A roadmap. In: CONFERENCE ON THE FUTURE OF SOFTWARE ENGINEERING. **Proceedings...** New York, NY, USA: ACM, 2000. p. 25–34.

GARNIER, S.; GAUTRAIS, J.; THERAULAZ, G. The biological principles of swarm intelligence. **Swarm Intelligence**, [S.l. : S.n.], v. 1, n. 1, p. 3–31, oct 2007.

GESTWICKI, P.; JAYARAMAN, B. Methodology and architecture of jive. In: ACM SYMPOSIUM ON SOFTWARE VISUALIZATION, 2005. **Proceedings...** New York, NY, USA: ACM, 2005. p. 95–104.

GRATI, H.; SAHRAOUI, H.; POULIN, P. Extracting sequence diagrams from execution traces using interactive visualization. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 17TH, 2010. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 2010. p. 87–96.

GROVE, D. et al. Call graph construction in object-oriented languages. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 32, n. 10, p. 108–124, oct. 1997. ISSN 0362-1340.

GU, Z. Capturing and exploiting fine-grained ide interactions. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 34TH, 2012. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2012. p. 1630–1631.

HIGHSMITH, J.; COCKBURN, A. Agile software development: the business of innovation. **Computer**, [S.l. : S.n.], v. 34, n. 9, p. 120–127, 2001.

HILL, E. **Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration**. Thesis (PhD), Newark, DE, USA, 2010.

HOFER, B.; WOTAWA, F. Combining slicing and constraint solving for better debugging: The conbas approach. **Adv. Soft. Eng.**, Hindawi Publishing Corp., New York, NY, United States, v. 2012, p. 13:13–13:13, jan. 2012. ISSN 1687-8655.

HÖLLDOBLER, B.; WILSON, E. **The Superorganism: The Beauty, Elegance, and Strangeness of Insect Societies**. [S.l.]: W.W. Norton, 2009.

IEEE. **IEEE Standard Glossary of Software Engineering Terminology**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990. 1 p.

INTERNATIONAL CONFERENCE ON BIG DATA AND SMART COMPUTING, 2015. Scalable graph exploration and visualization: Sensemaking challenges and opportunities. In: **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society, 2015. p. 271–278.

JAYARAMAN, S.; JAYARAMAN, B.; LESSA, D. Compact visualization of java program execution. **Software: Practice and Experience**, [S.l. : s.n.], p. n/a–n/a, 2016.

KATSO, H. Sdb: a symbolic debugger. In: UNIX PROGRAMMER'S MANUAL. [S.l.], 1979.

KATZ, I. R.; ANDERSON, J. R. Debugging: An analysis of bug-location strategies. **Hum.-Comput. Interact.**, L. Erlbaum Associates Inc., Hillsdale, NJ, USA, v. 3, n. 4, p. 351–399, dec. 1987.

KERSTEN, M.; MURPHY, G. C. Using task context to improve programmer productivity. In: ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 14TH, 2014. **Proceedings...** New York, NY, USA: ACM, 2006. p. 1–11.

KIM, S.; PAN, K.; WHITEHEAD, E. E. J. Memories of bug fixes. In: ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 14TH, 2006. **Proceedings...** New York, New York, USA: ACM Press, 2006. p. 35.

KNIGHT, C.; MUNRO, M. Should users inhabit visualizations? In: IEEE INTERNATIONAL WORKSHOPS ON ENABLING TECHNOLOGIES: INFRASTRUCTURE FOR COLLABORATIVE ENTERPRISES, 9TH, 2000. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2000. p. 43–50.

KO, A. Debugging by asking questions about program output. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 28TH, 2006. **Proceedings...** New York, NY, USA: ACM, 2006. p. 989–992.

KO, A. J.; MYERS, B. A. Finding causes of program output with the java whyline. In: PROCEEDINGS OF THE SIGCHI CONFERENCE ON HUMAN FACTORS IN COMPUTING SYSTEMS. **Proceedings...** New York, NY, USA: ACM, 2009. p. 1569–1578.

KUTTAL, S. K.; SARMA, A.; ROTHERMEL, G. Predator behavior in the wild web world of bugs: An information foraging theory perspective. In: 2013 IEEE SYMPOSIUM ON VISUAL LANGUAGES AND HUMAN CENTRIC COMPUTING. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 2013. p. 59–66.

LABICHE, Y.; KOLBAH, B.; MEHRFARD, H. Combining static and dynamic analyses to reverse-engineer scenario diagrams. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 29TH, 2013. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 2013. p. 130–139.

LATOZA, T. D.; MYERS, B. Developers ask reachability questions. In: ACM/IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 32ND, 2010. **Proceedings...** New York, New York, USA: ACM Press, 2010. v. 1, p. 185.

LAWRANCE, J.; BELLAMY, R.; BURNETT, M. Scents in programs: Does information foraging theory apply to program maintenance? In: IEEE SYMPOSIUM ON VISUAL LANGUAGES AND HUMAN-CENTRIC COMPUTING. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2007. p. 15–22.

LAWRANCE, J. et al. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In: CONFERENCE ON HUMAN FACTORS IN COMPUTING SYSTEMS, 26TH, 2008. **Proceedings...** New York, New York, USA: ACM Press, 2008. p. 1323.

LAWRANCE, J. et al. How programmers debug, revisited: An information foraging theory perspective. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 39, n. 2, p. 197–215, 2013.

LEE, S. et al. The impact of view histories on edit recommendations. **IEEE Transactions on Software Engineering**, IEEE Computer Society, Los Alamitos, CA, USA, v. 41, n. 3, p. 314–330, March 2015.

LI, L. et al. Chaos-order transition in foraging behavior of ants. **Proceedings of the National Academy of Sciences**, [S.l. : S.n.], v. 111, n. 23, p. 8392–8397, jun 2014.

LINTON, M. A. The evolution of dbx. In: SUMMER USENIX CONFERENCE. **Proceedings...** [S.l.], 1990. p. 211–220.

MAALEJ, W. et al. On the comprehension of program comprehension. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 23, n. 4, p. 31:1–31:37, sep. 2014.

MINELLI, R. et al. Visualizing developer interactions. In: IEEE WORKING CONFERENCE ON SOFTWARE VISUALIZATION, SECOND, 2014. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2014. (VISSOFT '14), p. 147–156.

MURPHY, G. C.; KERSTEN, M.; FINDLATER, L. How are java software developers using the eclipse ide? **IEEE Software**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 23, n. 4, p. 76–83, jul. 2006.

NGUYEN, T. T. et al. Recurring bug fixes in object-oriented programs. In: ACM/IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 32ND, 2010. **Proceedings...** New York, New York, USA: ACM Press, 2010. v. 1, p. 315.

OHMANN, P.; LIBLIT, B. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 28TH, 2013. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2013. p. 378–388.

OHMANN, P.; LIBLIT, B. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. **Automated Software Engineering**, [S.n.], [S.l.], p. 1–40, 2016.

P. Wainwright. **GNU DDD - Data Display Debugger**. [S.l.]: Free Software Foundation, 2010.

PARNIN, C.; ORSO, A. Are automated debugging techniques actually helping programmers? In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 2011. **Proceedings...** New York, NY, USA: ACM, 2011. p. 199–209.

PETRILLO, F. et al. Visualizing interactive and shared debugging sessions. In: **Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on**. Washington, DC, USA: IEEE Computer Society, 2015. p. 140–144.

PIORKOWSKI, D.; FLEMING, S. The whats and hows of programmers' foraging diets. In: CONFERENCE ON HUMAN FACTORS IN COMPUTING SYSTEMS, 2013. **Proceedings...** Paris, France: ACM, 2013. p. 3063–3072.

PIROLLI, P.; CARD, S. Information foraging. **Psychological Review**, [S.l. : s.n.], v. 106, n. 4, p. 643–675, 1999.

POTHIER, G.; TANTER Éric. Back to the future: Omniscient debugging. **IEEE Software**, IEEE Computer Society, Los Alamitos, CA, USA, v. 26, p. 78–85, 2009.

RESSIA, J.; BERGEL, A.; NIERSTRASZ, O. Object-centric debugging. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 34TH, 2012. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2012. p. 485–495.

ROMERO, P. et al. Debugging strategies and tactics in a multi-representation software environment. **Int. J. Hum.-Comput. Stud.**, Academic Press, Inc., Duluth, MN, USA, v. 65, n. 12, p. 992–1009, dec. 2007. ISSN 1071-5819.

RÖSSLER, J. How helpful are automated debugging tools? In: INTERNATIONAL WORKSHOP ON USER EVALUATION FOR SOFTWARE ENGINEERING RESEARCHERS, 1ST, 2012. **Proceedings...** [S.l.], 2012. p. 13–16.

SAITO, R. et al.

SALVANESCHI, G.; MEZINI, M. Debugging for reactive programming. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 38TH, 2016. **Proceedings...** New York, New York, USA: IEEE Computer Society Press, 2016. p. 796–807.

SANCHEZ, H.; ROBBES, R.; GONZALEZ, V. M. An empirical study of work fragmentation in software evolution tasks. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION, AND REENGINEERING, 22ND, 2015. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 2015. p. 251–260.

SILLITO, J.; MURPHY, G.; De Volder, K. Asking and Answering Questions during a Programming Change Task. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 34, n. 4, p. 434–451, jul 2008.

SOH, Z. et al. On the effect of program exploration on maintenance tasks. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 20TH, 2013. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 2013. p. 391–400.

STALLMAN, R. P.; SHEBS, S. **Debugging with GDB - The GNU Source-Level Debugger**. [S.l.]: GNU Press, 2002.

STOREY, M.-A. Theories, tools and research methods in program comprehension: past, present and future. **Software Quality Journal**, [S.l. : S.n.], v. 14, n. 3, p. 187–208, sep. 2006.

STOREY, M.-A. et al. The (r) evolution of social media in software engineering. In: THE ON FUTURE OF SOFTWARE ENGINEERING, 2014. **Proceedings...** New York, NY, USA: ACM, 2014. p. 100–116.

SUMPTER, D. J. T. The principles of collective animal behaviour. **Philosophical Transactions of the Royal Society B: Biological Sciences**, [S.l.], v. 361, n. 1465, p. 5–22, jan 2006.

TANENBAUM, A. S.; BENSON, W. H. The people's time sharing system. **Software: Practice and Experience**, [S.l. : s.n.], n. 2, p. 109–119, apr 1973.

TIARKS, R.; RÖHM, T. Challenges in Program Comprehension. **Softwaretechnik-Trends**, [S.l. : s.n.], v. 32, n. 2, p. 19–20, may 2012.

TSCHINKEL, W. R. The architecture of subterranean ant nests: beauty and mystery underfoot. **Journal of Bioeconomics**, Springer US, [S.l], v. 17, n. 3, p. 271–291, oct 2015.

WANG, S.; LO, D. Version history, similar report, and structure: Putting them together for improved bug localization. In: INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, 22ND, 2014. **Proceedings...** New York, NY, USA: ACM, 2014. p. 53–63.

WIKIPEDIA. **Collective intelligence**. 2015. Available from Internet: <http://en.wikipedia.org/wiki/Collective_intelligence>.

WIKIPEDIA. **Crowd**. [S.l.]: [S.n., 2016. Available from Internet: <http://en.wikipedia.org/wiki/Crowd>.

YE, X.; BUNESCU, R.; LIU, C. Learning to rank relevant files for bug reports using domain knowledge. In: ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 22ND, 2014. **Proceedings...** New York, NY, USA: ACM, 2014. (FSE 2014), p. 689–699.

YING, A. T.; ROBILLARD, M. P. The Influence of the Task on Programmer Behaviour. In: IEEE INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, 19TH, 2011. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society, 2011. p. 31–40.

ZAYOUR, I.; HAMDAR, A. A qualitative study on debugging under an enterprise ide. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 70, n. C, p. 130–139, feb. 2016. ISSN 0950-5849.

ZELLER, A. **Why programs fail: a guide to systematic debugging**. San Francisco, CA, USA: Elsevier Inc., 2006.

ZHENG, A. X. et al. Statistical debugging: Simultaneous identification of multiple bugs. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 23RD, 2006. **Proceedings...** New York, NY, USA: ACM, 2006. p. 1105–1112.

ZHOU, J.; ZHANG, H.; LO, D. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 34TH, 2012. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2012. p. 14–24.

ZIADI, T. et al. A fully dynamic approach to the reverse engineering of uml sequence diagrams. In: **IEEE International Conference on Engineering of Complex Computer Systems, 16th, 2011**. Los Alamitos, CA, USA: IEEE Computer Society Press, 2011. p. 107–116.

ZIMMERMANN, T. et al. Mining version histories to guide software changes. **IEEE Transactions on Software Engineering**, IEEE Computer Society, Washington, DC, USA, v. 31, n. 6, p. 429–445, June 2005.

ZOU, L.; GODFREY, M. W.; HASSAN, A. E. Detecting interaction coupling from task interaction histories. In: **Proceedings...** [S.l.: s.n.].

APPENDIX

Appendices 1 - Swarm Debugging - Interactive debugging experiment tutorial

QRS/ICSME 2016 - Experiment Tutorial

Questions? Please contact me: Fabio Petrillo (fabio@petrillo.com)

Thank you for accept this project. You have to follow this instructions to complete the tasks. Please, use Java 8 for running this experiment and follow theses steps:

1. Developer login

First, if you don't have a developer login, please ask us to receive a login.

2. Profile Survey

Please, fill this profile survey: http://goo.gl/forms/a7Is2DYGaI.

3. Screencast recording

During this work, all activities must be recorded using a screencast tool. Thus, when before to start any task (reading, searching, debugging), start to record your screen. When you complete a task, stop to record. You can use your prefered screencast tool, or Open Broadcaster Software (OBS). You can download this tool on http://obsproject.com.

To start a screencast recording (screen capture) using OBS follow this steps: - run the OBS - create a new Scene Collection - create a new Screen Capture to your screen - click "Start Recording" on right side.

This video explain how to do it: https://youtu.be/6s81E9l78yk.

4. Eclipse set up

You must use Eclipse Mars 2 for Java Developers for this work. If you already have this Eclipse version installed in your system, you can use it. However, we suggest to follow these steps:

1. create a new folder (swarmdbg for example)
2. download Eclipse Mars 2 for Java Developers, saving in your work folder. You can download on:
   - Linux 64-bits
   - Windows 64-bits
   - Other systems
3. extract it in the folderextract it in the folder
4. download the Swarm Debug Plugin on http://swarmdebugging.org/publications/experiment/SwarmManager_0.0.1.jar
5. move the plugin to /eclipse/plugins
6. run the installed Eclipse
7. choose your workspace
8. **setup Java Debug Step filtering**
   - click on menu Windows -> Preferences
   - search by "Step Filtering"
   - click on "Step Filtering"
   - select all checkboxes and click on "Ok"
9. open the Swarm Debugging Manager
   - click on menu Windows -> Show View -> Other
   - find "Swarm Debug" and select "Swarm Debug Manager"
   - click "OK"

Suggestion: organize this view on the left, below of Package Explorer.

In case of problem, please watch the video on https://youtu.be/pjxDNyK5nxs.

If you did not find a view "Swarm Debug" in the "Show View -> Other", probably you are using Java 7. Please install Java 8 or contact the experimenter for helping.

5. Warm up task

We propose a "warm up" task in order to test the environment setup and to learn some procedures to follow during the work. The task is simply use the Eclipse Debugger on a small Tetris game project. Departing from Game.handleStart method, arrive into the method Game.handleScoreModification() using "Step Into" (F5) and/or "Step Over" (F6).

Thus, please, follow these steps to complete the task:

1. download the warmup project from http://swarmdebugging.org/publications/experiment/tetris-1.2-src.zip
2. extract the Tetris in your workspace
3. run Eclipse and import the Tetris as a Java project
4. start to record a screencast (using OBS or any other tool)
5. open the Swarm Debugging Manager
    - click on menu Windows -> Show View -> Other
    - find "Swarm Debug" and select "Swarm Debug Manager"
    - click "OK"
6. login to Swarm Debug
    - click on Login button ("people" icon)
    - inform your login (username)
7. create a new Swarm Debug Session
    - in the Eclipse Package Explorer view or Project Explorer view, select the Tetris project (click it one time)
    - in the Swarm Debug Manager view, select the task "Tetris#001 Game.handle...."
    - finally, click on Create a session button (gear icon) to create a new session. Automatically, the stop red button will be activated.
8. execute the task using Eclipse Debugger toggle breakpoints start the Tetris using Run -> Debug stepping in the code until to arrive in the Game.handleScoreModification() method.
9. click on the stop session button to finish the session
10. stop the screencast recording

11. contact the experimenter to send your video

In case of problem, please watch the video on https://www.youtube.com/watch?v=U1sBMpfL2jc.

6. Experiment tasks

In this work, you will execute two defined tasks for you about JabRef project (http://www.jabref.org/). JabRef is an open source bibliography reference manager. The goal is to locate the bug, describing where (class(es) and method(s)) and why the issue happens in the system. For each task, you have one hour maximum to complete the task. If you can not locate the bug, please stop and fill the form resulting informing "Not found".

After start the task, please follow this steps:

1. download the JabRef project on http://swarmdebugging.org/publications/experiment/JabRef3.2.zip
2. extract it in your workspace
3. run in a console the command *./gradlew eclipse *
4. open the Eclipse and import the project in your workspace
5. start to record your screencast
6. remove all breakpoints
7. stop all active swarm sessions (if you have any)
8. execute a defined task, following the same steps of warm up task, but now for one of defined tasks for you. Double click on task will open the respective task description.
9. After finish, stop recording
10. send your video. You can use https://www.wetransfer.com, or any other system.
11. Finally, fill the form open the "Result Task Form" on http://goo.gl/forms/gworprqsPH. In this form, you will inform your results about the task.
12. execute from step 5 to 11 for the second defined task for you.

7. Payment

We will pay you by task, after analyse your artefacts and results. Thus, *you must *:

- start your recording

- clear all breakpoints in Eclipse
- stop any swarm session active
- start a new session
- analyse the issue/bug defined for you (double click to open the issue description)
- analyse the code
- toggle one or more breakpoints
- start the debugger to locate the bug
- after locate the bug and understand it (or not after one hour), stop the session and the video recording.
- fill the result form
- send the video.

After that, we will analyse your results and pay you if you follow correct that steps. In addition, we are available to clarify any question.

8. Thanking

Thank you so much for participating in this work. We will have several projects in the future and you will be invited if you show a good proactivity and attitude.

Appendices 2 - Publications

This thesis generated the following publications and therefore contains material from them:

1. **Fabio Petrillo**, Foutse Khomh, Marcelo Pimenta, Carla Freitas and Yann-Gaël Guéhéneuc, **Swarm Debugging: the Collective Debugging Intelligence of the Crowd**, IEEE Software (submitted).
2. **Fabio Petrillo**, Zéphyrin Soh, Foutse Khomh, Marcelo Pimenta, Carla Freitas and Yann-Gaël Guéhéneuc, **Towards Understanding Interactive Debugging**, Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), August 1-3, 2016, Vienna, Austria.
3. **Fabio Petrillo**, Zéphyrin Soh, Foutse Khomh, Marcelo Pimenta, Carla Freitas and Yann-Gaël Guéhéneuc, **Understanding Interactive Debugging with Swarm Debug Infrastructure**, Proceedings of the 24th International Conference on Program Comprehension (ICPC), May 16-17, 2016, Austin, Texas, USA.
4. **Fabio Petrillo**, Guilherme Lacerda, Marcelo Pimenta, and Carla Freitas, **Visualizing Interactive and Shared Debugging Sessions**, in Software Visualization - New Ideas and Emerging Results (VISSOFT/NIER), 2015 Third IEEE Working Conference on, 27-28 Sept. 2015, Bremen, Germany.
5. **Fabio Petrillo**, Guilherme Lacerda, Marcelo Pimenta, and Carla Freitas. **Swarm debugging: towards a shared debugging knowledge**. In III Workshop de Visualização, Evolução e Manutenção de Software (VEM), pages 65-72, 2015, Belo Horizonte, MG.
6. **Fabio Petrillo**, Guilherme Lacerda, Marcelo Pimenta, and Carla Freitas, **Polimorphic View: Visualizando o Uso de Polimorfismo em Projetos de Software.** In: 2nd Workshop on Software Visualization, Evolution and Maintenance, 2014, Maceió, AL. 2nd Workshop on Software Visualization, Evolution and Maintenance. Maceió, AL, 2014. v. U. p. 1-8.
7. **Fabio Petrillo**, Marcelo Pimenta, and Carla Freitas, **O Estado-da-Arte das Ferramentas de Visualização de Software.** In: XV Ibero-American Conference on Software Engineering (CibSE 2012), 2012, Buenos Aires. Proceedings of XV Ibero-American Conference on Software Engineering (CibSE 2012), 2012. v. U.

Appendices 3 - Highlighted comments from submitted paper reviews

In all, nine papers[1] were submitted for evaluation wich four were accepted (VEM 2015 and VISSOFT 2015 (NIER), ICPC 2016 (short paper) and QRS 2016) and four were rejected (VISSOFT (2015, 2016), ICSME 201 (ERA) and ICSME 2016). In this section are listed some encouraging comments on the work, organising by event and reviewer.

ICSME 2016

> *"The studied research question is both relevant and has not yet sufficiently covered in previous work, which the authors motivate well in the introduction and discussion of related work. The topic also clearly fits the scope of the conference. The suggested framework to collect the required data and the performed study are sound. I appreciate the efforts of the authors to involve professional developers and to investigate a realistic scenario. I believe the authors follow a promising direction and have already collected a valuable data set."*

> *"I started out being very intrigued and excited by the idea of this paper. I still think the work has great promise, but I find the current draft to be poorly written and not well argued. I would like to encourage you to work hard at repairing it, as I believe the results can be very important and influential. I like the overall idea of the paper of attempting to improve comprehension by investigating how developers place and work with breakpoints, and by sharing these information. Breakpoints can be a valuable data source given that they mark an explicit place in which a developer was interested during debugging."*

> *"I believe observing and learning how developers debug code and translating that knowledge into how we can assist them to perform debugging in a better way is very valuable and a motivated goal. As such I have no doubts about the motivation of the problem addressed in the paper."*

QRS 2016

> *"The idea to record and provide collective information for debugging purposes is quite nice."*

> *"It addresses an interesting problem."*

ICPC 2016

> *"Investigated an area that researchers/practitioners have relatively little insight into - debugging. Create an infrastructure that enables the study of debugging."*

> *'The paper addresses an important problem in program comprehension and software maintenance, namely the study and improvement of interactive debugging. The idea of swarm debugging and its knowledge sharing aspect is*

[1]The IEEE Software paper is under evaluation when we finished this thesis.

promising and worthwhile to study further. The SDI infrastructure provides an infrastructure for researchers to conduct debugging case studies and collect debugging data. The whole infrastructure is available on the web for other researchers to use."

"I think enabling data collection during debugging activities is very important and useful for understanding debugging activities. More importantly, the SDI enables this function without disturbing the debugging activities. That is, it makes SDI applicable. In addition, SDI also provides visualization of the collected data."

"The idea to support interactive debugging is also novel. The experiments in this paper are conducted on one real software, with 10 participants, and of which the results show some interesting observations."

VEM 2015

"Exciting research topic."

"The idea of joining code view with storage data from prior sessions or performed by different programmers is interesting. The implementation tool is a very positive point since managed to use several resources available on the Eclipse framework to develop the technique/tool."

"The paper covers a timely and relevant research topic of clear interest to the software maintenance and visualization communities."

ICSME 2015

"The idea itself is definitely promising and I would like to encourage more research in the area, but this work is still in a very early stage. The paper lacks any kind of evaluation. Instead, the authors report on some of their own experiences in using the tools."

VISSOFT 2015

"This also has a very good education component to it. Having students view exemplar debugging sessions by experts would greatly help. The search is an important part of the tool."

"An important problem is being addressed. Good use of collective intelligence."

"The authors identify a pain point in software engineering: debugging is a human activity performed individually by developers, and these developers accumulate knowledge that is either lost or simply not easily shared between developers on the same project. The paper is well organized. It motivates the work by identifying three major challenges in SE. The authors list several use cases for their tool - none have any empirical evaluations but this is to be expected of a NIER paper."

"I like the idea to look at debugging with the question if we can change it in a way that not all information is lost. Debugging right now is a completely "non persistant" activity. You can only learn with the person doing it, everything else is lost."

"This is a great idea to present for NIER, although some more work can be done on connecting with previous work (though not specific to debugging). Debugging is a specific and distinct enough activity that specific exploration and support of the topic is worth exploring."